An Introduction to Programming through C++

About the Author

Abhiram G Ranade received a BTech in Electrical Engineering from IIT Bombay in 1981 and a doctorate in Computer Science from Yale University in 1989. He was Assistant Professor of Electrical Engineering and Computer Science at the University of California, Berkeley, during 1988-95. Since 1995, he is on the faculty of IIT Bombay, currently as Professor of Computer Science and Engineering. He has received IIT Bombay *Excellence in Teaching* awards in 2006-2007 and 2010-2011.

An Introduction to Programming through C++

Abhiram Gorakhanath Ranade

Professor
Department of Computer Science and Engineering
Indian Institute of Technology (IIT) Bombay
Mumbai

McGraw Hill Education (India) Private Limited
NEW DELHI

McGraw Hill Education Offices
New Delhi New York St Louis San Francisco Auckland Bogotá Caracas
Kuala Lumpur Lisbon London Madrid Mexico City Milan Montreal
San Juan Santiago Singapore Sydney Tokyo Toronto

McGraw Hill Education (India) Private Limited

Published by McGraw Hill Education (India) Private Limited,
P-24, Green Park Extension, New Delhi 110 016.

An Introduction to Programming through C++

ISBN (13): 978-93-392-1886-7
ISBN (10): 93-392-1886-8

Managing Director: *Kaushik Bellani*

Head—Higher Education (Publishing and Marketing): *Vibha Mahajan*

Senior Publishing Manager (SEM & Tech. Ed.): *Shalini Jha*
Associate Sponsoring Editor: *Smruti Snigdha*
Senior Editorial Researcher: *Amiya Mahapatra*
Manager—Production Systems: *Satinder S Baveja*
Assistant Manager—Editorial Services: *Sohini Mukherjee*
Production Executive: *Anuj K Srivastava*

Assistant General Manager (Marketing)—Higher Education: *Vijay Sarathi*
Assistant Product Manager (SEM & Tech. Ed.): *Tina Jajoriya*
Senior Graphic Designer—Cover: *Meenu Raghav*

General Manager—Production: *Rajender P Ghansela*
Manager—Production: *Reji Kumar*

Typeset at Script Makers, 19, A1-B, DDA Market, Paschim Vihar, New Delhi, India

To my parents,
Shyama** and **Gorakhanath Ranade
who believed that everything is connected
and that everything can be understood

Contents

Preface

This book presents an introduction to programming using the language C++. No prior knowledge of programming is expected. However, the book is oriented towards a reader who has finished about 12 years of schooling in the Science stream. Our target readers are likely pursuing a degree in Science or Engineering, and only some will major in Computer Science.

We believe that our target reader will be able to learn programming by reading the book and solving suggested problems entirely through self-study. But the book can also be used as a text, and is detailed enough to serve as a reference.

Any programming textbook must accomplish three goals.

1. It must describe the syntax and semantics of the programming language.
2. It must explain the challenges in using the language to design programs.
3. Finally, the book must motivate the student to study for the first two goals!

The last goal requires explaining the position of programming in the world of science and engineering, and in the general world of ideas, and of course designing examples and exercises which excite the student. In the rest of this preface, we explain how we work towards these three goals.

Our approach, developed over several years of teaching programming to our target audience, has two unusual features.

1. First, the book is accompanied by a graphics package, Simplecpp. This package is useful for drawing and animating simple two-dimensional shapes. We believe this assists us in giving more vivid explanations of concepts as well as in assigning more interesting (and yet challenging) exercises to students.
2. Second, we have made a conscious effort to draw programming examples from a variety of areas, from math and science and even art. This has many advantages as will be seen shortly.

In Section 0.1, we begin by considering questions of motivation and philosophy. In Section 0.2, we outline our approach to teaching the C++ language. In Section 0.3, we discuss program design. In Section 0.4, we discuss the motivation behind our graphics package, Simplecpp.

We have also included some non-graphics features which we have found practically useful for getting students off to a quick start in writing interesting programs. This is considered in Section 0.5. Finally, in Section 0.6, we discuss how the book can be fitted into a curriculum involving one or two semesters.

0.1 The Philosophical Appeal of Computing

Science is what we understand well enough to explain to a computer.

Donald E Knuth

Why should anyone learn to write programs? A very persuasive answer is economic; programming is a skill which essentially guarantees you a job. Even if you do not plan to enter the software industry, programming is a useful auxiliary skill for many professions.

These considerations help in persuading students to study programming. But they will not necessarily make for excitement which is essential for learning. For this, it is necessary to explain why Computer Science and programming is intrinsically interesting.

It might be said that the goal of Science, or indeed of all human endeavour, is to understand the universe better. This understanding manifests itself as mental models of the various phenomena, and most such models are computational! Using a computer, you can make these mental models more explicit, and can more easily experiment with them. So computing is intimately connected to what it means to understand something. But this cannot be left as just another "nice" philosophical observation. It must be put into action.

The models you have of the world around you can be made to come alive on a computer. You have been told in Physics courses that the planets must move around the sun; but if you know programming, you can write a program to explore this claim. You can write programs to explore almost anything around you, transportation systems, chemical reactions, biological processes, social interactions. You can analyze pictures, or even literary texts, besides performing tax and salary calculations. Many of these exciting possibilities are within the grasp of our target audience. It is our experience that twelve years of school education is enough to enable students to raise questions about the world and also be able to answer some of those questions using programming skills learned in even a single course!

Thus, in this book we expect to not only teach programming, but also show how programming appears in a very diverse range of applications. We find that this appeals to our audience which doesn't intend to major in computer science, not at the beginning of the course anyway. We develop substantial programs for applications drawn from math, science, engineering, operations research, and even topics which are more like Art. Interestingly, many of these programs aptly illustrate important computer science concepts, e.g. recursion.

We thus believe that our treatment better integrates programming with the math and science skills (not to mention general worldly skills!) that the students already have. We feel this synergistically benefits the learning of computer programming and the other sciences.

Note, that by and large, we do not teach anything besides programming in this book. For building the applications we draw on what the student already knows. However, to make the book self-contained, we have included detailed descriptions of the concerned applications, highlighting the computational aspects.

The last few chapters contain somewhat advanced material, e.g. representations of graphs and their uses in representing circuits, the web graph, city maps. Also discussed are discrete event simulation, issues such as deadlocks in simulation, Dijkstra's shortest path algorithm as a simulation, a somewhat elaborate simulation of an airport, and finally the Newton-Raphson method in multiple dimensions with an application from mechanics. These are meant as help to students in doing projects and for exciting students to further study.

0.2 C++ Syntax and Semantics

This book presents the various constructs of C++ in a fairly comprehensive manner. We also present some of the more recent additions to the C++ language, such as lambda expressions. These are more convenient in many situations where previously function pointers or function objects were used. We dwell on different constructs in proportion to their importance; for example, arrays can be used in a variety of different ways: we provide examples of many such uses. In general, we attempt to provide a large number of examples for each programming construct. We also discuss some topics which might be considered "advanced", e.g. reference counting. This is discussed in Appendix G.

We have tried to present ideas in order of intellectual simplicity as well as simplicity of programming syntax. The general presentation style is: "Here is a problem which we would like to solve but we cannot using the programming primitives we have learned so far; so let us learn this new primitive". Object oriented programming is clearly important, but an attempt is made to let it evolve as a programming need. We discuss this below.

We also present a somewhat detailed overview of computer hardware. We feel that this is essential to satisfy the curiosity of our target audience, and also to make it easier for them to understand concepts such as program state, addresses and pointers, and also compilation.

C, C++ and Object Oriented Programming

The dominant paradigm in modern programming practice is clearly the object oriented paradigm. Likewise, C++ is clearly more convenient for the (experienced) programmer than C. So it could be asked: should we teach object oriented programming from "day 1"? Should we teach C++ directly or as an evolution of C?

Several educators have attempted to introduce classes and objects very early. But this is not considered easy, even by the proponents of the approach. The reasons are several. For example, for very simple programs, organizing programs into classes might be very artificial and verbose. Expecting a student to actually develop classes very early requires understanding function abstraction (for developing member functions/methods) even before control structures are understood. This can appear unmotivated and overwhelming.

Our discussion of object oriented programming can be considered to begin in Chapter 5: creating a graphical shape on the screen requires creating an object of a graphics class. In the initial chapters, it is only necessary to use classes, not build new classes. Thus, shapes can be created, and member functions invoked on them to move them around, etc.

The major discussion of classes including the modern motivations happens in Chapter 18. However, member functions are introduced in Section 17.5. Inheritance is presented in Chapter 25. Chapter 26 presents inheritance based design. It contains a detailed example in which a program developed earlier, without inheritance, is redeveloped, but this time using inheritance. This vividly shows how inheritance can help in writing reusable, extensible code.

A brief description of the use of inheritance in the design of Simplecpp graphics system is also given, along with an extension to handle composite objects.

As to the question of whether to teach C++ directly or C first, our view is pragmatic. As discussed earlier, most chapters begin with a ''crisis statement" which is followed by the resolution of the crisis in the rest of the chapter. We have attempted to order the crises in increasing order of intellectual complexity. Sometimes this causes C++ statements tobe introduced first, e.g. use of operators >> and

<< for input output rather than `scanf` and `printf`.[1] Sometimes C features get introduced first, which we then evolve to the C++ features. Thus, we present `structs` first (Chapter 17) and then generalize them to classes (Chapter 18). This enables us to gradually motivate each important idea. Likewise we discuss arrays before C++ vectors. This is because arrays are concrete and hence easier to understand. The abstraction of vectors must be studied, but it can be unconvincing at the beginning.[2]

We have a substantial discussion of the C++ standard library and template functions and template classes.

0.3 The Design of Programs

There is of course a distinction between learning the syntax and semantics of a programming language, and acquiring the ability to design programs. The former could be considered to be akin to program comprehension: someone who understands the syntax and semantics of a language should be able to work out how a program executes and predict what answers it will produce.

Design is very different. The phrase program design could mean one of two things: (a) writing a program when you reasonably well know what calculations are required to be performed, (b) first inventing the algorithm (usually requires some cleverness) and then coding it into a program. Interpretation (b) could perhaps be referred to as algorithm design. In this book, we consider algorithm design only to a small extent: specifically, we explore the use of recursion in designing algorithms. Some of the harder problems we consider here are backtrack search and structural recursion as applied to an expression drawing problem. We also discuss some analysis of algorithms, but we recognize that these are topics for later courses.

The major focus is on designing programs to solve problems, where the learner is generally conversant with how the problem is to be solved, i.e. how the problem could be solved manually using pencil and paper. There are many challenges in turning such informal knowledge into a program.

- Devising computer representations for real life/mathematical entities in the problem. Asserting invariant properties of the representations.
- Identifying the patterns in the computations that are required to be performed and expressing those computations using a programming language. This is difficult because the pattern in the computation may not directly match the primitives available in the language.
- Writing structured, extensible code. We discuss alternative ways of expressing the same logic, as well as issues relating to naming of variables, ideas such as avoiding use of global variables.
- Writing object oriented code. We have discussed this earlier.
- Reasoning about programs. We discuss ideas such as assertions, pre and post conditions, invariants, and use them in proving program correctness as appropriate.
- We discuss some elementary strategies and practices in testing and debugging.

Many of the challenges mentioned above arise naturally as we try to develop programs for problems drawn from science, math and other areas. Simple two-dimensional geometric graphics also provides fascinating programming problems, as we will discuss shortly.

[1] We do not discuss C language features such as `printf` and `scanf` which do not have any pedagogical merit. Likewise, we have omitted discussion of C language features such as unions because these are no longer relevant and are subsumed in inheritance.

[2] The later chapters of the book do use vectors and other standard classes as needed, rather than using arrays.

An important aspect of problems from science and other areas known to the student is that the student is generally familiar with the calculations that need to be performed. What is needed is deciding how to organize these calculations in general. Such problems are perfect for teaching programming without involving algorithm design.

0.4 Graphics and Simplecpp

I hear and I forget. I see and I remember. I do and I understand.

Confucius

A large part of the human experience deals with pictures and motion. Humans have evolved to have a good sense of geometry and geography, and are experts at seeing patterns in pictures and also planning motion. If this expertise can be brought into action while learning programming, it can make for a more mature interaction with the computer. It is for this reason that Simplecpp was primarily developed.

Our package Simplecpp contains a collection of graphics classes which allow simple geometrical shapes to be drawn and manipulated on the screen. Each shape on the screen can be commanded to move or rotate or scale as desired. Taking inspiration from the children's programming language Logo, each shape also has a pen, which may be used to trace a curve as the shape moves. The graphics classes enable several computational activities such as drawing interesting curves and patterns and performing animations together with computations such as collision detection. These activities are challenging and intuitive at the same time.

The graphics classes are used right from the first chapter. The introductory chapter begins with a program to draw polygons. The program statements command a turtle[3] holding a pen to trace the lines of the polygon. An immediate benefit is that this simple exercise makes the imperative aspect of programming and notions such as control flow very obvious. A more important realization, even from this very elementary session is the need to recognize patterns. A pattern in the picture often translates to an appropriate programming pattern, say iteration or recursion. Identifying and expressing patterns is a fundamental activity in programming in general. This principle is easily brought to the fore in picture drawing.

As you read along, you will see that graphics is useful for explaining many concepts, from variable scoping and parameter passing to inheritance based design. Graphical facilities make several traditional examples (e.g. fitting lines to points, or simulations) very vivid. Finally, graphics is a lot of fun, a factor not to be overlooked. After all, educators worldwide are concerned about dwindling student attention and how to attract students to academics.

0.5 First Day/First Month Blues

C++, like many professional programming languages, is not easy to introduce to novices. Many introductory programming books begin with a simple program that prints the message "hello world". On the face of it, this is a very natural beginning. However, even a simple program such as this appears

[3] Represented by a triangle on the screen, as in the language Logo.

complicated in C++ because it must be encased in a function, `main`, having a return type `int`. The notion of a return type is clearly inappropriate to explain on the very first day. Other concepts such as namespaces are even more daunting. The only available course of action is to tell the students, "don't worry about these, you must write these mantras whose meaning you will understand later". This doesn't seem pedagogically satisfactory.

After the student has somehow negotiated through `int`, `main` and namespaces, there is typically a long preparatory period in which substantial basic material such as data types and control structures has to be learnt, until any interesting program can be written. Psychologically and logistically, this "slow" period is a problem. Psychologically, a preparatory period without too much intellectual challenge can be viewed by the student as boring, which is a bad initial impression for the subject. Second, in most course offerings, students tend to have weekly lecture hours and weekly programming practice hours. In the initial weeks, students are fresh and raring to go. It is disappointing to them if there is nothing exciting to be done, not to mention the waste of time.

To counter these problems, the following features have been included in Simplecpp. Instead of the main program being specified inside a function main returning int, a preprocessor macro `main_program` is defined. It expands to "`int main()`". Thus, the main program can be written as

```
main_program{
body
}
```

Further, once the student loads in the Simplecpp package using `#include <simplecpp>`, nothing additional needs to be loaded, nor using directives given. The Simplecpp package itself loads other header files such as `iostream` and issues the using directives. These "training wheels" are taken off when functions, etc., are explained (Section 11.1).

A second "language extension" is the inclusion of a "repeat" statement. This statement has the form

```
repeat(count){
body
}
```

and it causes the body to execute as many times as the value of the expression count. This is also implemented using preprocessor macros and it expands into a for statement.

We believe that the repeat statement is very easy to learn, given a good context. Indeed, it is introduced in Chapter 1, where instead of using a separate statement to draw each edge of a polygon, a single line-drawing statement inside a repeat statement suffices. In fact, the turtle-based graphical drawing examples are compelling enough that students do not have any difficulty in understanding nested repeat statements either.

In the second and third chapters, there is a discussion of computer hardware, data representation and data types. These topics are important, but are not amenable to good programming exercises. For this period, the repeat statement together with the notions introduced in the first chapter can be used very fruitfully to generate relevant and interesting programming exercises.

0.6 Fitting the Book into a Curriculum

The book can be used for either a one-semester course or a two-semester sequence. For a one-semester course, the recommended syllabus is Chapter 1 through Chapter 7, Chapter 9 through Chapter 11, Chapter 13 through Chapter 15, Chapter 17, Chapter 18, Section 21.1, Chapter 22, and Chapter 25. Many of these chapters contain multiple examples of the same concept, all of these need not be "covered" in class. Some sections of these chapters could be considered "advanced", e.g. Section 10.4 which talks about game tree search for the game of Nim.

A two-semester sequence can cover Chapters 1 through Chapter 18 in the first semester, going over them carefully and considering at length aspects such as proving correctness of programs. The second semester could cover the remaining chapters. In the second semester, it would be appropriate to introduce (and use) some of the modern ideas such as reference counting pointers (Appendix G). A substantial programming project would also be appropriate. The book discusses many ideas for this.

Mumbai, May 2014 **Abhiram G Ranade**

Acknowledgements

It is a great pleasure to thank the many people who have directly or indirectly helped in writing this book.

Prof. D B Phatak introduced me to computer programming in a course he taught in my undergraduate years. This course was a moment of self-discovery for me. Since then Prof. Phatak has been a source of immense inspiration and energy. Even today his encouragement regarding this book is invaluable.

In my graduate years at Yale University, Nick Carrriero, David Gelernter and Paul Hudak created an ecosystem where programming issues were discussed with passion, if not fervour. It was great to share the excitement and the intensity.

In the Department of Computer Science and Engineering at IIT Bombay, it has been my privilege to discuss programming related topics with Om Damani, R K Joshi, Uday Khedker, Amitabha Sanyal, Soumen Chakrabarti, Parag Chaudhuri, S Sudarshan and others. These discussions have greatly helped in writing this book.

The initial drafts of the book were used in the introductory programming course at IIT Bombay by Soumen Chakrabarti, Sridhar Iyer, Supratim Biswas and me. I am grateful to them and also to the students whose response shaped the book.

Prachi Chawda read substantial portions of the book and solved problems in it. Individual chapters have been read and commented upon by Om Damani, Milind Sohoni, Uday Khedker, Subodh Gadgil, Devendra Bhave, Soumen Chakrabarti, and Siddhant Ranade. Discussions with K Ramchandran, A K Pani and Meghanad Ranade have helped with material in some chapters.

I would like to thank Devendra Bhave for the initial development of the package Simplecpp used in this book, and Satish Vemireddy for porting it to Windows.

I would also like to thank Lennart Johnsson, D M Dhamdhere and Umesh Vazirani who have influenced my overall outlook which perhaps is reflected in this book. I am also immensely grateful to Sandeep Bhat who tried to teach me how to write.

Last but not the least, I would like to thank my family, especially my wife, Hemala, without whose support this book would not have been possible.

Mumbai, May 2014 **Abhiram G Ranade**

Publisher's Note

The book is supplemented with separate online resources for instructors and students, accessible at *http://www.mhhe.com/ranade/cppp*.

The following is available:

- Lecture slides
- Graphics package Simplecpp
- Source codes for selected programs

The author and McGraw Hill Education (India) would like to thank all the reviewers for their help in shaping the nature of the book. Their names are given below:

Prabhat Verma	*Harcourt Butler Technological Institute (HBTI), Kanpur, Uttar Pradesh*
Preetvanti Singh	*Dayalbagh College of Engineering, Agra, Uttar Pradesh*
Subrajeet Mohapatra	*Birla Institute of Technology (BIT) Mesra, Ranchi, Jharkhand*
Prasanta Jana	*Indian School of Mines, Dhanbad, Jharkhand*
T V Gopal	*Anna University, Chennai, Tamil Nadu*
V Senthil Kumaran	*PSG College of Technology, Coimbatore, Tamil Nadu*

The author and McGraw Hill Education (India) invite suggestions and comments from you, all of which can be sent to *info.india@mheducation.com* (kindly mention the title and author name in the subject line).

Piracy-related issues may also be reported.

CHAPTER 1

Introduction

A computer is one of the most remarkable machines invented by man. Most other machines have a very narrow purpose. A watch shows time, a camera takes pictures, a truck carries goods from one point to another, an electron microscope shows magnified views of very small objects. Some of these machines are much larger than a computer, and many much more expensive, but a computer is much, much more complex and interesting in the kind of uses it can be put to. Indeed, many of these machines, from a watch to an electron microscope typically might contain a computer inside them, performing some of the most vital functions of each machine. The goal of this book is to explain how a computer can possibly be used for so many purposes, and many more.

Viewed one way, a computer is simply an electrical circuit; a giant, complex electrical circuit, but a circuit nevertheless. Computers have been made using mechanical gears, or fluidics devices,[1] but all that is mostly of historical importance. For practical purposes, today, it is fine to regard a computer as an electrical circuit. Parts of this circuit are capable of receiving data from the external world, remembering it so that it can be reproduced later, processing it, and sending the results back to the external world. By data we could mean different things. For example, it could mean some numbers you type from the keyboard of a computer. Or it could mean electrical signals a computer can receive from a sensor which senses temperature, pressure, light intensity and so on. The word process might mean something as simple as calculating the average of the sequence of numbers you type from the keyboard. It could also mean something much more complex, e.g. determining whether the signals received from a light sensor indicate that there is some movement in the vicinity of the sensor. Finally, by "send data to the external world" we might mean something as simple as printing the calculated average on the screen of your computer so that you can read it. Or we could mean activating a beeper connected to your computer if the movement detected is deemed suspicious. Exactly which of these actions happen is decided by a *program* fed to the computer.

It is the program which distinguishes a computer from most other machines; by installing different programs the same computer can be made to behave in dramatically different ways. How to develop these programs is the subject of this book. In this chapter, we will begin by seeing an example of a program. It turns out that we can understand, or even develop (typically called *write*) programs without knowing a lot about the specific circuits that the computer contains. Learning to write programs is

[1] Also it is appropriate to think of our own brain as a computer made out of biological material, i.e. neurons or neural cells.

somewhat similar to how one might learn to drive a car; clearly one can learn to drive without knowing how exactly an automobile engine works. Indeed, not only will you be able understand the program that we show you, but you will immediately be able to write some simple programs.

There are many languages using which programs can be written. The language we will use in this book is the C++ programming language, invented in the early 1980s by Bjarne Stroustrup. For the initial part of the book, we will not use the bare C++ language, but instead augment it with a package called *Simplecpp*. How to install this package is explained in Appendix A. This package was developed to make C++ appear more friendly and more fun to people who are starting to learn C++. To use the driving metaphor again, it could be said that C++ is like a complex racing car. When you are learning to drive, it is better to start with a simpler vehicle, in which there aren't too many confusing controls. Also, standard C++ does not by default contain the ability to draw pictures. The package Simplecpp does contain this feature. We thus expect that by using the Simplecpp package it will be easier and more fun to learn the language. But in a few chapters (by Section 11.6), you will outgrow Simplecpp and be able to use standard C++ (like "the pros"), unless of course you are using the graphics features.

1.1 A SIMPLE PROGRAM

Our first example program is given below.

```
#include <simplecpp>
main_program{
  turtleSim();

  forward(100);
  left(90);
  forward(100);
  left(90);
  forward(100);
  left(90);
  forward(100);

  wait(5);
}
```

If you *execute* this program on your computer, it will first open a window. Then a small triangle which we call a *turtle*[2] will appear in the window. Then the turtle will move and draw a square as it moves. After that, the window will vanish, and the program will end. First we will tell you why the program does all that it does. Then we will tell you how to execute the program.

The first line `#include <simplecpp>` declares that the program makes use of the Simplecpp package, in addition to what is provided by the C++ programming language.

The next line, `main_program{`, says that what follows is the main program.[3] The main program itself is contained in the braces `{ }` following the text `main_program`.

[2] Our turtle is meant to mimic the turtle in the Logo programming language.

[3] Yes, there can be non-main programs too, as you will see later. Also note that the phrase `main_program` is provided by Simplecpp. Specifying the main program is a bit more complex if you use bare C++; we will see this in Section 11.1.

The line following that, `turtleSim();` causes a window with a triangle at its center to be opened on the screen. The triangle represents our turtle, and the screen the ground on which it can move. Initially, the turtle points in the East direction. The turtle is equipped with a pen, which can either be raised or lowered to touch the ground. If the pen is lowered, then it draws on the ground as the turtle moves. Initially, the pen of the turtle is in the lowered state, and it is ready to draw.

The next line `forward(100)` causes the turtle to move forward by the amount given in the parentheses, `()`. The amount is to be given in *pixels*. As you might perhaps know, your screen is really an array of small dots, each of which can take on any colour. Typical screens have an array of about 1000 $\times$ 1000 dots. Each dot is called a pixel. So the command `forward(100)` causes the turtle to go forward in the current direction it is pointing by about a tenth of the screen size. Since the pen was down, this causes a line to be drawn.

The command `left(90)` causes the turtle to turn left by 90 degrees. Other numbers could also be specified instead of 90. After this, the next command is `forward(100)`, which causes the turtle to move forward by 100 pixels. Since the turtle is facing north this time, the line is drawn northward. This completes the second side of the square. The next `left(90)` command causes the turtle to turn again. The following `forward(100)` draws the third side. Then the turtle turns once more because of the third `left(90)` command, and the fourth `forward(100)` finally draws the fourth side and completes the square.

After this the line `wait(5)` causes the program to do nothing for 5 seconds. This is the time you have to admire the work of the turtle! After executing this line, the program halts.

Perhaps you are puzzled by the `()` following the command `turtleSim`. The explanation is simple. A command in C++ will typically require additional information to do its work, e.g. for the `forward` command, you need to specify a number denoting how far to move. It just so happens that `turtleSim` can do its work without additional information. Hence we need to simply write `()`. Later you will see that there can be commands which will need more than one pieces of information, in this case we simply put the pieces inside `()` separated by commas.

1.1.1 Executing the Program

To execute this program, we must first have it in a file on your computer. It is customary to use the suffix `.cpp` for files containing C++ programs. So let us suppose you have typed the program into a file called `square.cpp` – you can also get the file from the Simplecpp package.

Next, we must *compile* the file, i.e. translate it into a form which the computer understands more directly and can *execute*. The translation is done by the command `s++` which got installed when you installed the package Simplecpp. The command `s++` merely invokes an appropriate complier e.g. the GNU C++ compiler (See Appendix A). In a UNIX shell you can compile a file by typing `s++` followed by the name of the file. In this case you would type `s++ square.cpp`. As a result of this another file is produced, which contains the program in a form that is ready to execute. On UNIX, this file is typically called `a.out`. This file can be executed by typing its name to the shell

```
% a.out
```

You may be required to type `./a.out` because of some quirks of UNIX. Or you may be able to execute by double clicking its icon. When the program is thus executed, you should see a window come up, with the turtle which then draws the square.

Appendix A discusses how the compiler is to be invoked and how to execute the compiled program on systems besides UNIX.

1.2 REMARKS

A C++ program is similar in many ways to a paragraph written in English. A paragraph consists of sentences separated by full stops; a C++ program contains commands which must be separated by semicolons. Note that while most human beings will tolerate writing in which a full stop is missed, a computer is very fastidious, each command must be followed by a semicolon. Note however, that the computer is more forgiving about spaces and line breaks. It will accept spaces and linebreaks almost anywhere so long as words or numbers are not split. Thus, it is perfectly legal (though not recommended!) to write

```
turtleSim();forward(100)   ;
left  (90
);
```

if you wish. This flexibility is meant to enable you to write such that the program is easy to understand. Indeed, we have put empty lines in the program so as to help ourselves while reading it. Thus, the commands which actually draw the square are separated from other commands. Another important idea is to *indent*, i.e. put leading spaces before lines that are part of `main_program`. This is again done to make it visually apparent what is a part of the main program and what is not. As you might observe, indentation is also used in ordinary writing in English.

1.2.1 Execution Order

There is another important similarity between programs and text written in a natural language such as English. A paragraph is expected to be read from left to right, top to bottom. So is a program. By default a computer executes the commands left to right, top to bottom. But just as you have directives in magazines or newspaper such as "Please continue from page 13, column 4", the order in which the commands of a program are executed can be changed. We see an example next.

1.3 REPEATING A BLOCK OF COMMANDS

At this point you should be able to write a program to draw any regular polygon, say a decagon. You need to know how much to turn at each step. The amount by which you turn equals the exterior angle of the polygon. But we know from Euclidean Geometry that the exterior angles of a polygon add up to 360 degrees. A decagon has 10 exterior angles, and hence after drawing each side you must turn by $360/10 = 36$ degree. So to draw a decagon of side length 100, we repeat the `forward(100)` and `left(36)` commands 10 times. This works, but you may get bored writing down the same command several times. Indeed, you don't need to do that. Here is what you would write instead.

```
#include <simplecpp>
main_program{
  turtleSim();
  repeat(10){
        forward(100);
        left(36);
  }
  wait(5);
}
```

This program, when executed, will draw a decagon. The new statement in this is the `repeat` statement.[4] Its general form is

```
repeat(count){
  statements
}
```

In this, `count` could be any number. The `statements` could be any sequence of statements which would be executed as many times as the expression `count`, in the given order. The `statements` are said to constitute the *body* of the `repeat` statement. Each execution of the body is said to be an *iteration*. Only after the body of the loop is executed as many times as the value of `count`, do we execute the statement following the `repeat` statement.

So in this case, the sequence `forward(100); left(36);` is executed 10 times, drawing all 10 edges of the decagon. Only after that do we get to the statement `wait(5);`

1.3.1 Drawing Any Regular Polygon

Our next program when executed, asks the user to type in how many sides the polygon should have, and then draws the required polygon.

```
#include <simplecpp>
main_program{
  int nsides;
  cout << "Type in the number of sides: ";
  cin >> nsides;

  turtleSim();

  repeat(nsides){
        forward(50);
        left(360.0/nsides);
  }
  wait(5);
}
```

This program has a number of new ideas. The first statement in the main program is `int nsides;` which does several things. The first word `int` is short for "integer", and it asks that a region be reserved in memory in which integer values will be stored during execution. Second, it gives the name `nsides` to the region and stipulates that from now on, whenever the programmer uses the name `nsides` it should be considered to refer to this region. It is customary to say that `nsides` is a *variable*, whose *value* is stored in the associated region of memory. This statement is said to *define* the variable `nsides`. As many variables as you want can be defined, either by giving separate definition statements, or by writing out the names with commas in between. For example, `int nsides, length;` would

[4] The `repeat` statement is not a part of C++ but is provided by Simplecpp. C++ does have statements using which we can achieve repetition, but these are more involved and will be introduced in Chapter 7. The `repeat` statement is inspired by the Logo programming language.

define two variables, the first called `nsides`, the second `length`. We will learn more about names and variables in Chapter 3.

The next new statement is relatively simple. `cout` is a name that refers to the computer screen. It is customary to pronounce the `c` in `cout` (and `cin` in the next statement) as "see". The sequence of characters `<<` denotes the operation of writing something on the screen. What gets written is to be specified after the `<<`. So the statement in our program will display the message

Type in the number of sides:

on the screen. Of course, you may put in a different message in your program, and that will get displayed.

In the statement after that, `cin >> nsides;`, the name `cin` refers to the keyboard. It asks the computer to wait until the user types in something from the keyboard, and whatever is typed is placed into the (region associated with the) variable `nsides`. The user must type in an integer value and then press the return (sometimes called "Enter") key. The value typed in gets placed in `nsides`.

You may wish to note that the `>>` and `<<` operators are suggestive of the direction in which information flows.

After the `cin >> nsides;` statement is executed, the computer executes the `repeat` statement. Executing a repeat statement is nothing but executing its body as many times as specified. In this case, the computer is asked to execute the body `nsides` times. So if the user had typed in 15 in response to the message asking for the number of sides to be typed, then the variable `nsides` would have got the value 15, and the loop body would be executed 15 times. The loop body consists of the two statements `forward(100)` and `left(360.0/nsides)`. Notice that instead of directly giving the number of degrees to turn, we have given an expression. This is allowed! The computer will evaluate the expression, and use that value. Thus, in this case the computer will divide 360.0 by the value of the variable `nsides`, and the result is the turning angle. Thus, if `nsides` is 15, the turning angle will be 24. So it should be clear that in this case, a 15-sided polygon would be drawn.

1.3.2 Repeat Within a Repeat

What do you think the program below does?

```
#include <simplecpp>
main_program{
  int nsides;

  turtleSim();

  repeat(10){
    cout << "Type in the number of sides: ";
    cin >> nsides;
    repeat(nsides){
        forward(50);
        left(360.0/nsides);
    }
  }
  wait(5);
}
```

The key new idea in this program is the appearance of a `repeat` statement inside another `repeat` statement. How does a computer execute this? Its rule is simple: to execute a repeat statement, it just executes the body as many times as specified. In each iteration of the outer repeat statement there will be one execution of the inner repeat statement. But one execution of the inner repeat could have several iterations. Thus, in this case a single iteration of the outer repeat will cause the user to be asked for the number of sides, after the user types in the number, the required number of edges will be drawn by the inner repeat statement. After that, the next iteration of the outer repeat would begin, for a total of 10 iterations. Thus, a total of 10 polygons would be drawn, one on top of another.

1.4 SOME USEFUL TURTLE COMMANDS

The following commands can also be used.

`right(angle)`: This causes the turtle to turn right by the specified angle, which must be in degrees.

`penUp()`: This causes the pen to be raised. So after executing this command, the turtle will move but no line will be drawn until the pen is lowered. There is nothing inside the () because no number is needed to be specified, as was the case with `forward`, e.g. `forward(10)`.

`penDown()`: This causes the pen to be lowered. So after executing this command, a line will be drawn whenever the turtle moves, until the pen is raised again.

Thus, if you write `repeat(10){forward(10);penUp();forward(5);penDown();}` a dashed line will be drawn.

1.5 NUMERICAL FUNCTIONS

The commands you have seen so far for controlling the turtle will enable you to draw several interesting figures. However, you will notice that it is cumbersome to draw some simple figures. For example, if you wish to draw an isoceles right-angled triangle, then you will need to take square roots—and we haven't said how to do that. Say you want to draw a simple right-angled triangle with side lengths in the proportion 3:4:5. To specify the angles would require a trigonometric calculation. We now provide commands for these and some common operations that you might need. You may wonder, how does a computer calculate the value of the sine of an angle, or the square root of a number? The answers to these questions will come later. For now, you can just use the following commands without worrying about how the calculation actually happens.

Let us start with square roots. If you want to find the square root of a number `x`, then the command for that is `sqrt`. You simply write `sqrt(x)` in your program and during execution, the square root of `x` will be calculated, and will be used in place of the command. So for example, here is how you can draw an isoceles right-angled triangle.

```
forward(100);
left(90);
forward(100);
left(135);
forward(100*sqrt(2));
```

The commands for computing trigonometric ratios are `sine, cosine` and `tangent`. Each of these take a single argument: the angle in degrees. So for example, writing `tangent(45)` will be as good as writing 1.

The commands for inverse trigonometric ratios are `arcsine`, `arccosine` and `arctangent`. These will take a single number as an argument and will return an angle (in degrees). For example, `arccosine(0.5)` will be 60 as expected. These commands return the angle in the range -90 to +90. An important additional command is `arctangent2`. This needs two arguments, `y` and `x`, respectively. Writing `arctangent2(y,x)` will return the inverse tangent of `y/x` in the full range, -180 to +180.

To draw a triangle with side lengths 75, 100, 125, you may simply execute the following.

```
forward(75);
left(90);
forward(100);
left(arctangent2(75,-100));
forward(125);
```

As you might guess, we can put expressions into arguments of commands, and put the commands themselves into other expressions, and so on.

Some other useful commands that are also provided are the following:

1. `exp, log, log10`: These return respectively for argument x the value of e^x (where e is Euler's number, the base of the natural logarithm), the natural logarithm and the logarithm to base 10.
2. `pow`: This takes 2 arguments, `pow(x,y)` returns x^y.
3. `sin, cos, tan` respectively return the sine, cosine, and tangent of an angle, but it must be specified in radians.
4. `asin, acos, atan2` respectively return the arcsine, arccosine and arctangent, in radians. The command `atan2` takes 2 arguments `x, y` like the command `arctangent2` discussed above, and returns the inverse tangent of `y/x` in the range $-\pi$ to π.

The name `PI` can be used in your programs to denote π, the ratio of the circumference of a circle to its diameter.[5]

1.6 COMMENTS

The primary function of a program is to get executed. So it must be written following the rules described above.

However, a program should also be written so that it is easy to understand, when programmers read it. The reason is simple. One programmer may write a program, which another programmer may need to modify. In such cases, the second programmer must be able to understand why the program was written in the manner it was written. This process can be aided if the original programmer writes additional notes to explain the tricky ideas in the program. For this purpose, C++ allows you to insert

[5] The name `PI` as also the commands `sine, cosine, tangent, arcsine, arccosine` and `arctangent` are part of Simplecpp. All others e.g. `sqrt, pow` and the trigonometric commands using radians are a part of C++.

comments in your program. A comment is text that is not meant to be executed, but is meant solely for humans who might read the program.

A comment can be written in two ways. At any point on a line, you may place two slash characters, `//`, and then the text following the `//` to the end of the line becomes a comment. Alternatively, anything following the `/*` characters becomes a comment, and this comment can span over several lines, ending only with the appearance of the characters `*/`.

It is customary to put comments at the beginning mentioning the author of the program and stating what the program does. Subsequently, wherever something non-obvious is being done in the program, it is considered polite to explain using a comment.

Here is our polygon-drawing program written the way it should be written.

```
#include <simplecpp>
/* Program to draw a regular polygon with as many sides as the
   user wants.
   Author: Abhiram Ranade
   Date:   18 Feb 2013.
*/
main_program{
  int nsides;
  cout << "Type in the number of sides: ";
  cin >> nsides;

  turtleSim();

  repeat(nsides){
     forward(50);         // Each side will have length 50 pixels.
     left(360.0/nsides); // Because sum(exterior angles of a
                          // polygon) = 360.
  }
  wait(5);
}
```

1.7 COMPUTATION WITHOUT GRAPHICS

Although we began this introduction with a picture-drawing program, every program you write need not contain any drawing. Here is a program that does not draw anything, but merely reads a number from the keyboard, and prints out its cube.

```
main_program{
  int n;
  cout <<"Type the number you want cubed: ";
  cin >> n;
  cout << n*n*n << endl;
}
```

After `cout <<`, we are expected to place what we want printed. Here, we have written `n*n*n`, which causes the cube of `n` to be calculated and then printed. Instead of `n*n*n`, we could have written whatever mathematical expression we want. We discuss the exact rules later, in Section 3.2.

1.8 CONCLUDING REMARKS

Although it may not seem like it, in this chapter you have already learned a lot.

First, you have some idea of what a computer program is and how it executes: starting at the top and moving down one statement at a time going towards the bottom. If there are `repeat` statements, the program executes the body of the loop several times; the program is said to *loop* through the body for the required number of iterations.

You have learned the notion of a variable, i.e. a region of memory into which you can store a value, which can later be used while performing computations.

You have also learned several commands using which you can draw, do calculations (e.g. take square root).

You have also seen the notion of *generalization*: the polygon drawing program we wrote in Section 1.3.1 is a generalization of the square drawing program of Section 1.1. Generalization is very important; usually you will write programs which will not do just one fixed task, but behave differently depending upon the input given by the user. For this you need to be able to understand the principle behind whatever it is that you are doing, and express it in a general form as we did in the polygon drawing program.

A very important point concerns observing the *patterns* in whatever you are doing. When we draw a polygon, we repeat the same action several times. This is a pattern that we can mirror in our program by using the `repeat` statement. By using a repeat statement we can keep our program compact; indeed we may be drawing a polygon with 100 sides, but our program only has a few statements. You will see other ways of capturing patterns in your programs later. In general this is a very important idea.

Last but not the least, it is worth noting that we have tried to make our programs look good and be easy to understand. The main reason for this is that when we write programs, they will not only be executed on a computer, but will also be read by other programmers who must understand what we mean. We used indentation and comments for making our program more understandable. Later on in the book we will see other ways.

At this point, you should also see why the notation used to write programs is called a *language*. A natural language (e.g. English) is very flexible and general. It has a grammatical structure, e.g. there is a subject, verb, and object; or there can be clauses, which can themselves contain subjects, verbs, objects and other clauses. This organization is also present in programming languages; for example, you have already seen that a repeat statement can contain another repeat statement inside. There are also rules of punctuation, e.g. each statement of C++ must end with a semicolon, just as each sentence in the English language ends with a full stops. These grammatical and punctuation rules are often denoted by the term *syntax*. This term is also used in programming.

Our treatment of the C++ programming language will be somewhat similar to how you might be taught a new natural language, say Marathi or French. We will teach you the grammar, or the syntax, but that will be just one aspect. In order to be able to speak or write a language well, you must not only know the grammar and the vocabulary, but also know how to organize your thoughts and express yourself. Likewise, when learning C++, you must not only know the various statements and their syntax, you must understand how to *design* programs. Further, when you learn a new natural

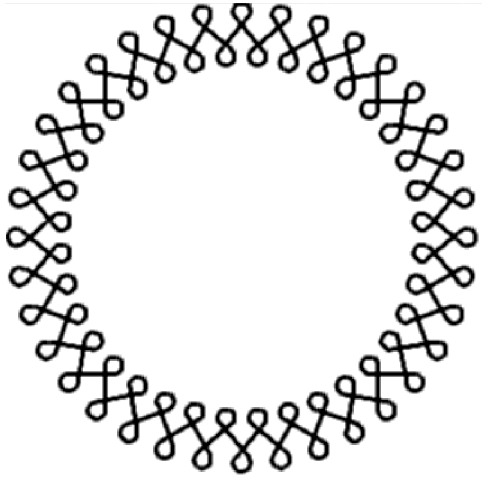


Fig. 1.1 *Can you draw this?*

language, you will typically read interesting literature in that language. The analogue while learning a programming language is: *you must learn how interesting computational problems get solved*! We will follow this strategy. Hope you will find it enjoyable.

1.8.1 Graphics

The main activity that computers engage in is of course calculating with numbers. However, there are many reasons we began this introduction with picture drawing, and why picture drawing will be an important parallel theme that will run through the book.

A program is not merely a list of calculations you want done; as you will see it is important to understand the patterns in the calculation and represent them in your program. Both these activities: understanding patterns and representing them in your program, are needed also when you draw pictures. In general, both the activities are quite difficult. But in case of pictures, the patterns are often very obvious. Thus, you can focus your attention on the task of representing them in the program. We will see many examples of this later.

Also note that drawing interesting pictures requires much careful calculation, using geometry and trigonometry that you have learned earlier.[6] Thus, picture drawing will provide another domain in which you can practice your computational skills.

Also remember the adage "A picture is worth a thousand words.". Indeed, it is very useful if you can show the result of your computation through a picture. Also, in a lot of applications it is useful if you can provide input to the program by drawing a picture or clicking on the screen, rather than by typing in numbers. In general, and especially for computation on mobile phones and tablet computers, the areas of *data visualization* and *graphical user interfaces* are becoming very important, and our picture drawing exercises will give you a taste of these areas.

And finally, drawing pictures is fun.

[6] Do not worry if you have forgotten some of this; we will refresh your memory when needed.

1.8.2 A Note Regarding the Exercises

Programming is not a spectator sport. To really understand programming, you must write many, many programs yourself. That is when you will discover whether you have truly understood what is said in the book. To this end, we have provided many exercises at the end of each chapter, which you should assiduously solve.

Another important suggestion: while reading many times you may find yourself asking, "What if we write this program differently". While the author will not be present to answer your questions, there is an easy way to find out—write it differently and run it on your computer! This is the best way to learn.

EXERCISES

In all the problems related to drawing, you are expected to identify the patterns/repetitions in what is asked, and use `repeat` statements to write a concise program as possible. You should also avoid excessive movement of the turtle and tracing over what has already been drawn.

1. Modify the program given in the text so that it asks for the side length of the polygon to be drawn in addition to asking for the number of sides.
2. Draw a sequence of 10 squares, one to the left of another.
3. Draw a chessboard, i.e. a square of side length, say 80, divided into 64 squares each of side length 10.
4. If you draw a polygon with a large number of sides, say 100, then it will look essentially like a circle. In fact this is how circles are drawn: as a many sided polygon. Use this idea to draw the numeral 8 – two circles placed tangentially one above the other.
5. A pentagram is a five pointed star, drawn without lifting the pen. Specifically, let A,B,C,D,E be 5 equidistant points on a circle, then this is the figure A–C–E–B–D–A. Draw this.
6. Draw a seven-pointed star in the same spirit as above. Note however that there are more than one possible stars. An easy way to figure out the turning angle: how many times does the turtle turn around itself as it draws?
7. We wrote "360.0" in our program rather than just "360". There is a reason for this which we will discuss later. But you could have some fun figuring it out. Rewrite the program using just "360" and see what happens. A more direct way is to put in statements `cout << 360/11; cout << 360.0/11;` and see what is printed on the screen. This is an important idea: if you are curious about "what would happen if I wrote ... instead of ...?" – you should simply try it out!
8. Read in the lengths of the sides of a triangle and draw the triangle. You will need to know and use trigonometry for solving this.
9. When you hold a set of cards in your hand, you usually arrange them fanned out. Say you start with cards stacked one on top of the other. Then you rotate the ith card from the top by an amount proportional to i (say $10i$ degrees to the left) around the bottom left corner. Now, we can see the top card completely, but the other cards are seen only partially. This is the figure that you are to draw. (a) Draw it assuming the cards are transparent. (b) Draw it assuming the cards are opaque. For (b), some trigonometric calculation will be necessary. In both cases, use `repeat` statements to keep your program small as possible.

10. Draw a pattern consisting of 7 circles of equal radius: one in the center and 6 around it, each outer circle touching the central circle and two others. Try to write a program which minimizes turtle movement. Your program statements should be chosen to exploit the symmetry in the pattern.

11. Draw the picture shown in Figure 1.1. As you can see, the picture has 36 repetitions of a basic pattern. Your program should be able to take a number n as input, and draw a pictures having n repetitions. Make sure that the lines and the arcs in the pattern connect smoothly.

12. Write a program which reads in a number and prints its fourth power.

13. Write a program which does the following 10 times: read in a number and then print its square root.

CHAPTER 2

A Bird's Eye View

When you begin a long journey, it is useful and reassuring to have an idea of where you are going, how difficult the journey is going to be, and of course how enjoyable. You may have similar questions and apprehensions when you start studying a new subject. In this chapter, our goal is to anticipate and answer some such questions.

1. How can a single machine like a computer solve problems from so many different areas like searching through documents, predicting the weather, making timetables for trains, or playing chess, not to mention calculating taxes?
2. Should you think of programming as a new activity, different from everything you have done so far? Or is it similar to things you have learned in school?
3. How does a computer work? What does it mean to say that it solves problems? Do we need to know about circuits in order to be able to write computer programs?

In this chapter, we give a very high level answer to the first two questions. They are discussed in great detail in the rest of the book. The third question we discuss at some length. How to design computers is not the subject of this book; our goal is to provide you with a mental model of a computer to the extent it is needed for learning to program.

In Section 2.1, we take up the first question. Basically, to solve any problem on a computer, you must first formulate it as a problem on numbers. We discuss this with examples. Once a problem is expressed numerically, you try to figure out what operations must be performed on the numbers in order to solve the problem. In this step, you can pretend, if you wish, that you are solving the problem manually using pencil and paper. Next you write a program, which instructs the computer to perform the required calculations, instead of you doing it on paper.

In Section 2.2, we discuss what programming is. In simple terms, a program is nothing but a very precise description the calculations needed to solve a problem. Two important points should be noted about this: while a computer may perform millions of calculations, your program does not have to mention each calculation separately: you can put the calculations in `repeat` like statements. Second, there is a tendency to consider computers and computing to be a modern activity, but in fact human beings have been doing calculations, and some sophisticated calculations at that, for thousands of years. You have yourself been doing several calculations very systematically in high school. In writing

a program, you need to describe those calculations systematically and precisely. It is fair to say that writing programs is a refinement of a skill that you already possess.

In the last part of the chapter, we consider questions such as how computers store numbers and perform calculations on them. This discussion is very elementary, and at a very high level. This discussion has three goals: (a) to satisfy your curiosity about what is inside a computer, (b) assure you that a computer has been designed in some sense to mimic the way humans organize computation, and so you should expect to find its working intuitive, (c) familiarize you with some simple ideas which you will directly use later in the book. In Section 2.3, we discuss the basic features of the circuits used in computers. In Section 2.4, we discuss in detail different formats used for representing numbers inside computers. In Section 2.5, we discuss the overall organization of a computer. Then we consider how individual parts work. We discuss the concept of a program stored in memory, and other concepts relevant to programming such as the concept of an *address*. We conclude by discussing what it means to compile a C++ program. The chapter contains a lot of detail which is given only for the purpose of illustrating the ideas. The details should not be interpreted literally, or too carefully remembered.

2.1 PROBLEM SOLVING USING COMPUTERS

As discussed above, the first step in solving a problem using a computer is to express it as a problem on numbers. This is easy for several real-life problems which are represented numerically to begin with. Commerce requires us to keep track of prices and profits and capital and salaries, and clearly this requires numbers and substantial computation on those numbers. Numbers are also obviously needed to represent quantities such as temperature, length, mass, force, voltage, concentration of chemicals. So it would seem that problems involving such quantities will be naturally formulated using numbers. However, it is not clear that this holds for all real-life entities. For example, can we express pictures or language using numbers? We discuss these questions next.

Here is how a picture might be represented using numbers. Consider a black and white picture to begin with. We first divide the picture into small squares by putting down a fine grid over it, as in Figure 2.1(a). Then for each small square, we determine whether it is more white or more black. If the square is more white we assign it the number 0, if it is more black, we assign it the number 1. So if we have divided the picture into $m \times n$ small squares (*pixels*), m along the height and n along the width, we have a sequence of mn numbers, each either 1 or 0 that represents the picture. Figure 2.1 shows the numbers we have assigned to each square. Given the mn number representation, we can reconstruct the picture as follows: wherever a 0 appears, we leave the corresponding square white, wherever a 1 appears, we make the corresponding square black. The reconstruction, using the numbers in Figure 2.1(b) is shown in Figure 2.1(c). As you can see, the reconstructed picture is not identical to the original picture, but reasonably similar. By choosing a finer grid, we would have been able to get a better approximation of the original picture. It turns out that pixels of size about 0.1 mm are good enough, i.e. the reconstructed picture is hard to distinguish from the original because our eye cannot individually see such fine squares. *Processing* a picture means doing computations involving these numbers. For example, changing every zero to a one and vice versa, will change the picture from "positive" to "negative"! Similar ideas are used when we wish to display pictures on a computer monitor, as will be discussed in Section 2.8.2.

It should be noted that the idea of putting down a grid over the object of interest is very powerful. Suppose we wish to represent the worldwide weather. So we divide the surface of the globe into small regions. For each region we consider the current state, i.e. parameters relevant to the weather such as

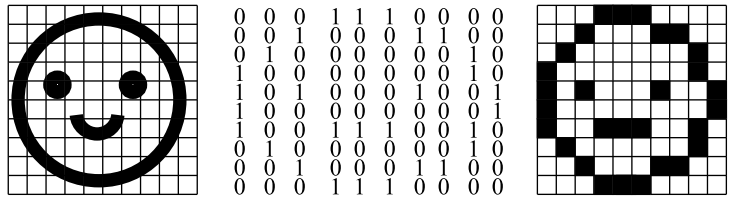


Fig. 2.1 *A picture, its representation, and reconstruction*

the ambient temperature, pressure, humidity. Of course, all points in a region will not have identical temperature but we nevertheless can choose an approximate representative temperature, if the region is reasonably small. And similarly, pressure or humidity. This collection of state information for all regions is a representation of the current worldwide weather. Given the current state of a region and the laws of physics we can calculate what the next state will be only by looking at the state of the nearby regions. This is a very gross simplification of how the weather is predicted, but, it is correct in essence.[1]

Text is represented using numbers as follows. Essentially, we device a suitable code. The most common code is the so called ASCII (American Standard Code for Information Interchange) code. For example, in the ASCII code, the letter "a" is represented as the number 97, "b" as 98, and so on. Punctuation marks and standard symbols as also the *space* character have a numerical code assigned to them. So the word "computer" is represented by the sequence of numbers 99,111,109,112,117,116,101,114. Sentences and paragraphs are also represented as number sequences, each letter, spaces, and punctuation mark represented by its numerical code from the ASCII code. Finding whether a given word occurs in a given paragraph is simply checking whether one sequence of numbers is a subsequence of another sequence of numbers! Note that the ASCII code is used to represent all text written using the Roman alphabet, including the C++ programs you will write. The *Unicode Consortium* provides codes to represent text in other alphabets, such as Devanagari.

We will see more real-life objects (and mathematical objects too, such as sets, functions) and how to represent them in the rest of the book.

2.2 ALGORITHMS AND PROGRAMS

After a problem has been represented numerically, the next step is to solve it. For this, we need to decide what operations to perform and in what order. Such a sequence of operations, described precisely, is said to constitute an *algorithm*. In deciding what operations to perform, you don't really need to consider a computer. To a good extent, you might as well be thinking about doing the calculations by hand, on paper.

An algorithm can have steps of the form "Multiply these two numbers, then add the result to the ratio of these other two numbers," and so on. You can also have steps such as, "If this number is zero then do this". Or also something like "Keep on doing this until ...". The key requirement is that there should be no ambiguity about what is to be done at any point in the algorithm. Once you have determined the algorithm, i.e. the precise sequence of actions, you can think about expressing it in C++. That will give

[1] This is not to say that all physical phenomenon related to the weather are well understood. In fact, many simple things are not understood, e.g. how precisely do raindrops form. However, we understand enough (through the hard work of several scientists) to make predictions with some confidence. The specific calculations are of course well outside the scope of this book.

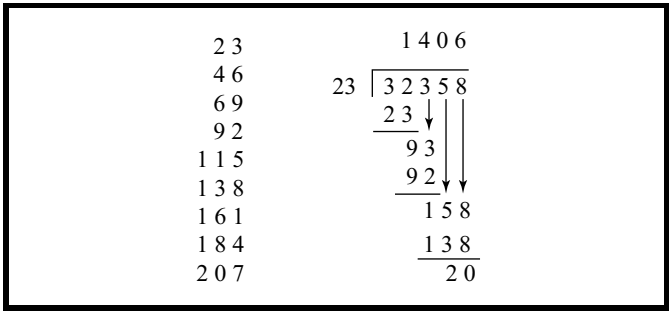


Fig. 2.2 *Primary-school division algorithm*

us the program. Of course, for doing the last step you need to know C++. This you will learn in the rest of the book.

We should point out that while the term *algorithm* may be new to you, you have actually learned many algorithms starting from primary school. For example, you know how to determine whether a given positive integer n is prime. Probably, you will do this as follows: starting from the integer 2, try out all integers till $n-1$ and check if they divide n (without leaving a remainder). If you find an integer that divides n, then declare n to be composite. If you don't find any such integer, then declare n to be a prime. This is indeed an algorithm to determine whether a number n is prime! We will soon see (Section 6.7.3) how it can be turned into a C++ program. You have learned algorithms even earlier. For example, you learned how to add up numbers, or subtract or multiply or divide them. These procedures are also algorithms! You probably learned these procedures by example, or through pictures, such as the one in Figure 2.2, in which you are first asked to make a multiplication table for the given divisor (in this case 23), and then on the right you actually perform the division. Even these basic algorithms will be of value. Indeed, they will come in handy when you want to perform arithmetic with very large numbers, as in Exercise 19 of Chapter 14.

It may interest you to know that many important computer programs of today are based on algorithms that were invented long ago. In Section 8.4, you will study an algorithm for finding the square root of a number which was used by the Babylonians, some 3500 years ago. Another example is Euclid's algorithm for finding the greatest common divisor of two numbers (Section 7.7). The notion of putting a grid over the surface of the earth for the purpose of predicting the weather also predates modern electronic computers. You might also perhaps know that many ancient civilizations did sophisticated calculations in order to determine the trajectories of the stars and the planets. Such ideas have lead to the very sophisticated programs of today, and we will see some of this in Chapter 19.

For the most part in the book, we will consider program design rather than algorithm design. By this we mean that we will assume that it is clear to you what calculations need to be performed; the question of interest would be how to express those calculations as C++ programs. For this, we will draw problems from domains that are familiar to you, e.g. high school/junior college mathematics and science, and of course every day life. How to write programs that are easy to understand, easy to use, easy to modify if need be, will be important concerns which you will appreciate only later in the book. We will discuss creative algorithm design only to a small extent: we will consider several examples of *recursion* (Chapter 10) which is a basic algorithm design technique.

2.3 BASIC PRINCIPLES OF DIGITAL CIRCUITS

We begin our discussion of how computers work by looking at the circuits used in computers. Here is a key fact: the circuits in a computer are designed such that for practical purposes, we can pretend that numbers flow through the wires in the circuit, or get stored in the devices in the circuit. Such circuits are called *digital circuits*. We only discuss digital circuits in this chapter. In digital circuits, at any time instant, we can think of each individual wire as carrying a single number, or to be more precise, either the number 0 or the number 1. Likewise, there are devices, that are capable of performing a storage function (most commonly capacitors), and each such individual device can also store the number 0 or the number 1 at any time.

We briefly explain how this illusion is created. But you can ignore this paragraph if you wish, it is not needed for understanding the rest of the book. As you may know, current flows through the wires in an electrical circuit (just as water flows through pipes), and wires are associated with voltages (electrical equivalent of water pressure). The idea for representing numbers in circuits is simple: if a wire is at a certain designated high voltage (say higher than 1 volt) then we will say that the number 1 is being carried on it. If the wire is at a certain designated low voltage (say smaller than 0.2 volts), then we will say that the wire is carrying the number 0. Note further that the circuits are designed so that the wires never carry voltages in the range 0.2 volts to 1 volt, and so there is never any ambiguity. Thus we can pretend that wires in the circuit are carrying around numbers. Further note that if you store electrical charge on a capacitor, the charge does not dissipate quickly; in this sense the capacitor remembers that charge. To make the capacitor remember a 0, we simply drain off charge from it. This will happens if we connect the capacitor to our designated low voltage. If on the other hand, we connect our capacitor to a high voltage, a large amount of charge gets stored on it; this represents the number 1. For the rest of the book, we will not worry about charges and voltages. Instead we will only talk about capacitors and wires holding and carrying the numbers 0 or 1.

Of course, we will want to store or communicate numbers besides 0 and 1. We will see how to do this in Section 2.4. Once we have numbers represented, it is possible to design circuits which can perform arithmetic on them. This is considered in Section 2.7.

2.4 NUMBER-REPRESENTATION FORMATS

The term *bit* is used to denote a number which is either 0 or 1, so we will say that each wire in a computer can carry a single bit, or each capacitor can store a single bit. If we want to represent other numbers, we can do so by associating with them a sequence of bits. As an example, say we decide to associate the sequence of bits 11001 with the number 25. Then whenever we want to store 25, we will need to use 5 capacitors, and in them store the respective bits of the sequence, i.e. 1, 1, 0, 0, 1. Likewise, if we want to send the number 25 from one device to another, we must have 5 wires, and on those we must respectively send 1, 1, 0, 0, 1. The question then is, what bit sequence should we associate with each number?

The simplest idea, discussed in detail in Section 2.4.1, is as follows: we represent a number using the sequence of bits given by its binary representation. So as an example, suppose we wish to represent the number 25. It has binary representation 11001. Thus it would be represented by the sequence of bits 11001, as discussed above. This idea is fine if we only wish to represent non-negative integers.

But our program may deal with integers which can be either positive or negative e.g. temperature rounded to the nearest degree. Thus we need a more complex scheme to represent such numbers. This is discussed in Section 2.4.2.

More generally, we may have to represent quantities such as mass, force, and velocities, which in general will be real numbers, and may be either positive or negative. Schemes for representing real numbers are discussed in Section 2.4.3.

There is one more issue to consider. In the example above, we said that the number 25 could be represented by a sequence of 5 bits. On most computers, the standard representation schemes require you to choose the length of the sequence to be one of 8, 16, 32, or 64 bits. This is because restricting the size of the bit sequence to these values makes it easy to design the circuitry in the computer. Note that it is customary to use the terms byte, half-word, word, and double-word to respectively mean 8, 16, 32 and 64 bits.

2.4.1 Unsigned Integer Representation

Suppose we know that a certain quantity we deal with in our program will always be a non-negative integer, e.g. a telephone number. In that case, as discussed above, we can represent it using the sequence of bits given by its binary representation. As mentioned above, the length of the representation must be chosen to be one of 8, 16, 32, 64. Thus if the number we wish to represent has a shorter binary representation than the length we chose, then we simply make the more significant bits 0, e.g. if we wish to represent 25 using 32 bits, the representation will be the bit string

$$00000000000000000000000000011001$$

Note that if we decide to use an n-bit long binary representation, the maximum value that can be represented is $2^n - 1$ (sequence of n 1s). Thus, we must be sure that the number we wish to represent is not larger. As an example, since we know that telephone numbers (in India) are at most 8 digits long, and since the largest possible 8-digit number $99999999 < 2^{32} - 1$, we can use $n = 32$ to represent telephone numbers.

2.4.2 Signed Integers

An integer can be negative or positive. So this throws a challenge: how do we represent negative numbers?

The simplest representation is the so called sign-magnitude representation. In this, if we have n bits to be used for representing the number, one of these is designated as a *sign bit*. We will set this bit to 0 if the number is positive, and to 1 if the number is negative. The remaining $n - 1$ bits will be used for representing the absolute value of the number. We might use the bit in the most significant position as the sign bit, so the representation for -25 using 32 bits would be

$$10000000000000000000000000011001$$

Notice that since we have decided to use $n - 1$ bits to represent the magnitude, the magnitude can be at most $2^{n-1} - 1$ (all $n - 1$ bits must be 1s). Since the numbers can be positive or negative, using n bits total we can represent numbers between $-2^{n-1} + 1$ and $2^{n-1} - 1$, both inclusive.

A more commonly used representation is the so-called 2's complement representation. The n bit 2s complement representation is defined as follows. In this the integer x is represented by the n-bit binary representation of the number x if $0 \leq x \leq 2^{n-1} - 1$, and by the n-bit binary representation of the number $2^n - x$ if $-2^{n-1} \leq x < 0$. Numbers outside this range cannot be represented.[2]

[2] This only means that there is no standard, built-in mechanism for representing numbers outside the range. However, you will be able to design your own mechanisms if you wish, as you are asked to in Exercise 19 of Chapter 14.

Here is how -25 would be represented in 32-bit 2's complement representation. Since -25 is negative, we have to represent it by the binary number $2^{32} - 25 = 4294967296 - 25 = 4294967271$. Then we take its binary representation, which is[3]

$$11111111111111111111111111100111$$

Thus, if you want to store -25 or send it on some wires, the above bit pattern will have to be stored or sent.

2.4.3 Floating Point Representations

Much computing needs to be done with real numbers. For example, velocities of particles, voltages, temperatures and so on in general need not take only integral values. Real numbers are represented using the so-called floating point representations. Usually, floating point representations use bit strings of length 32 or 64.

In the scientific world, real numbers are typically written using the so-called *scientific notation*, in the form: $f \times 10^q$, where the *significand* f typically has a magnitude between 1 and 10, and the *exponent* q is a positive or negative integer. For example the mass of an electron is 9.109382×10^{-31} kilograms, or Avogadro's number is 6.022×10^{23}.

On a computer, real numbers are represented using a binary analogue of the scientific notation.[4] So to represent Avogadro's number, we first express it in binary. This is not hard to do: it is

$$1.11111110001010101111111 \times 2^{1001110}$$

Note that this is approximate, and correct only to 3 decimal digits. But then, 6.022×10^{23} was only correct to 3 digits anyway. The exponent 1001110 in decimal is 78. Thus the number when written out fully will have 78 bits. We could use 78 bits to represent the number, however, it seems unnecessary. Usually, we will not need that much precision in our calculations. A better alternative, is to represent each number in two parts: one part being the significand, and the other being the exponent.

For example, we could use 8 bits to represent the exponent, and 24 bits to represent the significand, so that the number is neatly fitted into a single 32-bit word! This turns out to be essentially the method of choice on modern computers. You might ask why use an 8-24 split of the 32 bits and why not 10-22? The answer to this is: experience. For many calculations it appears that an exponent of 8 bits is adequate, while 24 bits of precision in the significand is needed. There are schemes that use a 64 bit double word as well and the split here is 11-53, again based on experience.

Note that the significand as well as the exponent can be both positive or negative. One simple way to deal with this is to use a sign-magnitude representation, i.e. dedicate one bit from each field for the sign. Note that we don't need to explicitly store the decimal point (or we should say, binary point!)—it is always after the first bit of the significand. Assuming that the exponent is stored in the more significant part or the word, Avogadro's number would then be represented as

$$0, 1001110, 0, 11111111000101010111111$$

[3] You may find it convenient to first convert 2^{32} and 25 to binary, and then subtract.

[4] In the scientific notation, the position of the decimal point within the significand depends upon the value of the exponent. Hence, the name *floating* point.

Two points are to be noted: (a) we have put commas after the sign bit of the exponent, the exponent itself, and the sign bit of the significand, only so it is easy to read. There are no commas in memory. (b) Only the most significant 23 bits of the significand are taken. This requires throwing out less significant bits (what happened in this example), but you might even have to pad the significand with 0s if it happens to be smaller than 23 bits.

As another example, consider representing -12.3125. This is -1100.0101 in binary, i.e. 1.1000101×2^3. Noting that our number is negative and our exponent is positive, the representation would be

$$0, 0000011, 1, 11000101000000000000000$$

Again, the commas are added only for ease of reading.

The exact format in which real numbers are represented on modern computer hardware and in C++ is the IEEE Floating Point Standard. It is much more complicated, but has more features, some of which we will discuss later.

2.5 ORGANIZATION OF A COMPUTER

We can think of a computer as consisting of the following main parts. An actual computer will contain more parts, but all are not important in this high-level sketch.

1. **Main memory**. In this, we store the numbers on which we are performing our calculations. As we will see later, the memory will also hold the program.
2. **Arithmetic unit**. This is capable of performing arithmetic. We supply to it the operands, tell it what operation we want performed, and it does so. We can then extract the result and store it back in memory.
3. **Input-output devices**. There can be many, but we consider the keyboard, the display, which is often referred to as the monitor or the screen, and the disk.
4. **Control unit**. This controls the other units, as the name implies.
5. **Network**. This is useful for moving data between the parts.

It is customary to use the term *Central Processing Unit* to denote the control unit together with the arithmetic unit.

You may think of each part as consisting of a box with circuitry inside. Each part has *ports* (sets of wires) on which data can come out from the part or go into the part. It is possible to take the data out of one part and send it to another part through the network. How exactly the data flows is determined by the control unit. This organization is sketched in Figure 2.3. The control unit has connections to every other unit, we have not shown them in the picture to avoid clutter.

2.6 MAIN MEMORY

The memory of a modern computer may contain a huge number of basic memory elements, typically a power of two, say 2^{35}. These are usually capacitors as discussed earlier. Each basic memory element is capable of storing 1 bit. The number of bits that can be stored is defined to be the capacity, or the size, of the memory. More commonly, the memory size is measured in bytes, where a byte is simply 8 bits. The terms kilo, mega, giga are used to respectively denote $2^{10} = 1024$, $2^{20} = 1048576$, $2^{30} = 1073741824$. As you can see these numbers are reasonably close to the metric equivalents, i.e. 1000, 1000000, 1000000000. Analogously, the terms tera, peta and so on are also defined.

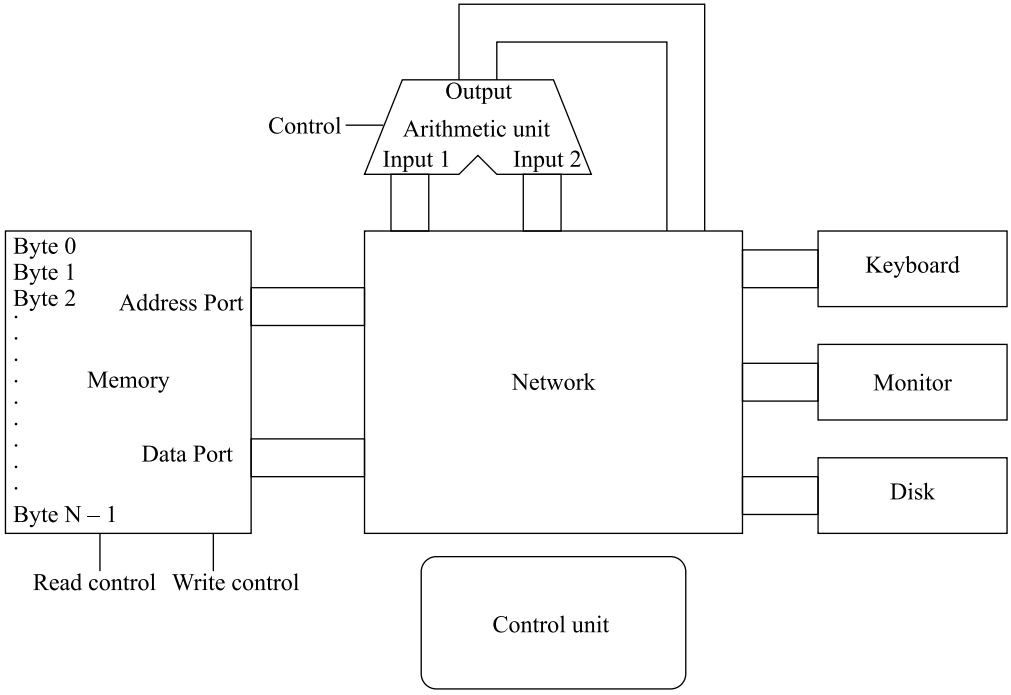


Fig. 2.3 *Computer organization (sketch)*

Thus, a memory with 2^{35} capacitors has size 2^{35} bits or 2^{32} bytes, or 4 Gigabytes.

2.6.1 Addresses

Each byte (group of 8 elementary memory devices, say capacitors) in the memory is associated with a unique label, or *address*. If a memory has N bytes, then the addresses can start at 0 and end at $N-1$ (Figure 2.3). Note that while in day to day life we would have used labels from 1 to N, on computers it is more customary to start with 0. The address of a byte is useful for identifying it from among all the bytes in the memory. Note that addresses are unsigned integers. Thus, they can be represented using their binary representation.

The phrase "byte x" is commonly used to mean the byte whose address is x. The phrase "word x" is also used, this simply means the word starting at byte x, i.e. the set of bytes $x, x+1, x+2, x+3$. Similarly, for half-words and double words.

The phrase "location x" is also used; usually, it means the word starting at address x. However, it may mean byte, halfword, or double word starting at x based on the context.

2.6.2 Ports and Operations

A memory communicates with the rest of the world using 2 sets of wires or ports. The first is the address, and the second the data port. There are also two additional wires connecting to the memory: we will call the first the read control port and the second the write control port. Using these, we can access the contents of the memory as follows. In what follows we use the phrase "place a quantity" to mean "place the representation of that quantity".

❑ Storing Data into Byte x

For this, it is necessary to place the address x on the address port, and the data that you want stored, say the number y, on the data port. Then you place the number 1 on the write control port. This signals the memory to store the data on the data port into the byte whose address is present on the address port. Thus, the number y will be stored in byte x. Byte x will continue to hold the number y until another write operation is performed on byte x.

❑ Reading Data from Byte x

For this, you place the address x, on the address port. Then you place a 1 on the read control port. The 1 on the read control port signals the memory to sense the data stored in byte x of the memory and place it on the data port. Once data appears on the data port, it can be moved from there to where it is needed.

What we described above is a *byte-oriented* memory. More common are *word-oriented* memories. In these, when we supply an address, the word starting at the given address is sent back or written to. In byte-oriented memories, the data port will consist of 8 wires, because 8 bits need to be communicated. In word-oriented memories, the data port will likewise have to have 32 wires. Similarly, for half-words and double-word oriented memories.

How many wires do we need in the address port? Let us take our 2^{32} byte memory as an example. In this memory, the addresses range from 0 to $2^{32} - 1$. Thus the largest address consists of 32 consecutive 1s. Hence, the address port will have to have 32 wires, in order that we may specify any possible address. In general, if the memory has N bytes, then we will need to have $\log_2 N$ wires in the address port.

2.7 THE ARITHMETIC UNIT

The arithmetic unit has circuits using which it is possible to perform basic arithmetic operations, i.e. addition, subtraction, multiplication, division, for numbers in all formats described earlier, unsigned and signed integers, and floating. It receives the operands through two ports named Input1 and Input2 and the result of the operation is placed on the port named Output, see Figure 2.3. What operation is to be performed depends upon the value supplied on the Control port. The arithmetic unit can also convert numbers from one representation to another, e.g. given a number represented as an integer on one of the inputs, its representation in the floating format (exponent and significand) can be produced on the output port.

You may think that the arithmetic unit must consist of many very complicated circuits. That is indeed true. However, for the purpose of programming, we don't need to know how the circuits are to be designed, it is sufficient to know what they can do.

2.8 INPUT-OUTPUT DEVICES

The input-output devices are considered to be *peripherals*, and the rest of the computer the "main computer".

2.8.1 Keyboard

The simplest input device is a keyboard. A code number is assigned to each key on the keyboard. When a key is pressed, the corresponding code number is sent to the main computer. The control unit decides what is to be done with the received code number; for example it might just get stored in the memory.

2.8.2 Display

A computer terminal screen or display is a fairly complex device. You probably know that a display is made up of *pixels* which are arranged in a grid, say 1024 rows and 1024 columns. Each pixel can be made to show the colour you desire. By showing appropriate colours, you can display pictures, or letters, or the turtle from Chapter 1. The display hardware decides what colour to show in a pixel by consulting a small amount of memory associated with each pixel. The amount of memory depends on the sophistication of the display. For a simple black and white display, it is enough to specify whether the pixel is to appear white or black. So a single bit of memory is enough. You may also have displays which can show different levels of brightness: k bits of memory will be able to store numbers between 0 and $2^k - 1$ and hence that many levels of brightness, or gray levels. In colour displays we need to simultaneously store the red, green, blue components at each pixel, and so presumably even more bits are needed. Indeed, high quality colour displays might use as many as 24 bits of memory for each pixel. To display an image, all we need to do is to store appropriate values in the memory associated with each pixel in the screen. If we have 24 bits of memory per pixel, then because there are $1024 \times 1024 = 2^{20}$ pixels, we will need a memory with addresses between 0 and $2^{20} - 1$, each cell of the memory consisting of 24 bits. A reasonable correspondence is used to relate the pixels and addresses in memory: the colour information for the pixel (i, j) i.e. the pixel in row i and column j (with $0 \leq i, j < 1024$) is stored in address $1024i + j$ of the memory. When the circuitry of the screen needs to display the colour at pixel (i, j) it picks up the colour information from address $1024i + j$ of the memory. If you wish to change the image, it suffices to changes the data in the memory. So in some ways, the display can be treated very much like another memory. The main computer can access this memory, often called the *display memory* or *video memory*. should not be confused with the main memory of the computer.

2.8.3 Disks

Devices such as disks can also be thought of as storing data at certain addresses; however, the addresses no longer refer to specific capacitors in the circuitry, but specific regions on the surface of the disk. The surface can be magnetized in different directions: the direction indicates whether a 0 or a 1 is stored there.

Optical compact disks also function in a similar manner. The surface of an optical compact disk has elevations and dips which can be detected by shining a laser on them. Whether a certain region of the disk stores a 0 or it stores a 1 is determined by the pattern of elevations and dips in that region.

2.8.4 Remarks

There is a lot of innovation and ingenuity in designing peripheral devices. This is of course outside the scope of this book.

However, the fundamental ideas should be noted: (a) communication between the main computer and the peripheral device happens by sending numbers, (b) information is stored as bits, by designating

some physical property to determine whether a 0 or a 1 is stored, and (c) if several bits are stored, there will be a notion of *address* using which we can refer to some selected bit or group of bits.

2.9 THE CONTROL UNIT

As the name implies, the Control Unit controls the other parts of a computer. It behaves like a factory manager who tells the workers what to do and when. The control unit may at one step ask the arithmetic unit to perform addition, while at another step it might ask the memory to supply the data from a certain address inside it. At another step it might ask the network to send the data from the memory to the arithmetic unit and so on.

Clearly, the control unit must command the rest of the computer to perform the computation needed by the program being executed on the computer. For this, the control unit must effectively be given a suitable version of your program. This version is somehow placed in the memory of the computer, and then the control unit can then fetch it from memory, a little bit at a time, and get the computer parts to take the required steps. We describe the process of converting your program into a version that the control unit can understand in Section 2.11. But don't panic; most of this happens smoothly without much effort from you. The description in this section is only to tell you what happens "behind the scenes".

The control unit can only "understand" programs written in the so called *machine language* of the computer. This is simply a numerical code language devised by the designer of the computer. Just as in the ASCII code each character is represented by a number, in the machine language each operation that the computer is capable of is represented by a number. For example, the designer may choose to represent the operation of reading two 32 bit unsigned integers from memory, taking their product and storing it back into memory by some number, say 57. More specifically, the computer designer might state that the sequence $57, x, y, z$ represents the operation of taking 32 bit unsigned integers stored (in the words starting) at addresses x, y taking their product and storing the product back (in the word starting at) at address z. Such sequences will have to be designated for every operation that can conceivably happen on the computer. It is customary to call each such a sequence an *instruction*. A machine-language program is simply a sequence of such instructions which cause the desired computation to happen.

We will explain this with an example. Suppose we wish to compute the cube of a number, let us call it x. Suppose x is already in memory, in the word at address 100. Suppose now the control unit is asked to execute the instruction

$$57, 100, 100, 104$$

As described above, this would cause the computer to fetch the number stored at location 100, twice, and feed the two copies of the number to the arithmetic unit. Then the control unit would command the arithmetic unit to perform multiplication. After this, the control unit would cause the product to be stored back into memory at location 104. Thus the square of x will be computed and stored in location 104. Note that location 100 continues to hold x. To get the cube we must get the control unit to perform another multiplication using the instruction:

$$57, 100, 104, 104$$

This instruction would cause the number in locations 100 and 104 (i.e. x and x^2) to be sent to the arithmetic unit and multiplied, and the product would be stored back into location 104. Thus, at the end of the two instructions, location 104 would hold x^3.

The sequence of numbers 57, 100, 100, 104, 57, 100, 104, 104 would thus be the *machine language program* to cube a number. Of course, in a useful program the number to be cubed would have to be first read from the keyboard, and at the end the cube would have to be displayed on the screen. To perform each of these operations, several additional instructions will be needed. The machine language program would then be the concatenation of all such numeric sequences. This large sequence would then have to be loaded into the memory of the computer, and the control unit asked to execute the instructions in the sequence.

You may wonder whether the machine language contains an analogue of the `repeat` command discussed in Section 1.3. Indeed, there will have to be. Some sequence of numbers would have to be designated by the computer designer to mean "go back and execute this sequence of instructions one more time". Designing the machine language is a very tricky blend of science and art. We will not discuss it further; but we hope that the above discussion has given you a rough idea of how a program executes.

2.10 THE TIMING MODEL

So far we have not discussed the time required to perform each operation described above. For example, how long does it take to multiply two numbers? Exact answers to these questions are very tricky, and very involved. Part of the reason is that on a real computer, there are many other complications. However, a very simplistic, but reasonably useful answer can be given.

The answer is as follows. With every computer, we can associate a *clock rate* or a *clock speed* which indicates how fast the circuits in the computer can perform operations. The clock speed is measured as a frequency, number of operations per second. At the time of writing this, a typical clock speed for computers is 3 GHz, which means that the computer is capable of performing 3 billion operations per second. By operation, we typically mean arithmetic operations, i.e. add, subtract, multiply, and divide. It is also acceptable to assume, to keep things simple, that data can be fetched from memory in about the same time.

What does this imply as far as programming is concerned? Simply that whatever problem we may be solving, we should try to finish it using as few operations as possible. If we can do this, our program will run fast. You will see that the same problem can be solved in many ways, some of them requiring fewer operations than others. Exercise 6 at the end of this chapter gives you a taste of this. So one challenge in programming is to figure out how to minimize the number of operations performed. Of course, the primary challenge is to find some way of solving the problem correctly in the first place.

Note by the way that peripheral devices typically do not operate as fast as the arithmetic unit and the memory. For most problems considered in this book, we will not have much choice in how to access peripheral devices, be they disk or the display or the keyboard. So we will not consider this issue.

2.11 HIGH-LEVEL PROGRAMMING LANGUAGES

When the earliest computers were built, they could be used only by writing machine language programs. Indeed, you had to decide where in memory you would store your data, look up the computer manual and determine what instruction would perform the actions you wanted, and then write out the sequence of numbers that would constitute the machine language program. Then the machine-language program would have to be loaded into the computer memory, and then you could execute the program. As you might guess, this whole process is very tiring and error prone.

Fortunately, today, programs can be written in the style seen in Chapter 1, and to be discussed in the rest of the book. We do not think about what instructions to use, nor the address in memory where to store the number to be cubed or the number of sides of the polygon we wish to draw. Instead, we use familiar mathematical expressions to denote operations we want performed. We give names to regions of memory and store data in them by referring to those names. The computer, of course, really only "understands" instruction codes and memory addresses, and does not understand mathematical notation or the names we give to parts of memory. So how does our nice looking program actually execute on a computer?

Clearly, the nice-looking programs we write must first be translated into machine language instructions which the computer does understand. This is done by a program called a *compiler*, which fortunately has been written by someone already! The program `s++` that you used in the last chapter is a C++ compiler, which takes a C++ program (e.g. the one from Section 1.7) and generates the file (e.g. a.out) which contains a machine language program like what we discussed in Section 2.9. When you type

```
a.out
```

from the command line or click on the program icon, the content of the file `a.out` gets loaded into the memory, and then what is loaded starts getting executed.

2.12 CONCLUDING REMARKS

For the purpose of learning to program, it is useful to summarize what we have learned.

In the first part of the chapter, we noted that the information/data which we wish to process should be represented as a collection of numbers. You should formulate a representation scheme using which the problem to be solved is represented as a set of numbers, and the desired answer is also represented as a set of numbers. The program must then be given the first set of numbers and it must perform computation on that set and generate the second set. Some problems are naturally represented as numerical problems, whereas for others, we must use some kind of a numerical coding scheme, like the ASCII coding scheme. We also defined the notion of algorithms, and gave examples of algorithms that are learned in primary school.

In the second half of the chapter, we noted that numbers are themselves represented by electrical signals on a computer, i.e. as voltages. There are (memory) circuits which can store these voltages, and other circuits (arithmetic unit) which process one set of voltages and generate another set of voltages. For example, every computer will have a circuit to which you can supply voltages representing the numbers to be added together, and such that the circuit will produce voltages representing the sum of the numbers. We also saw some specific number representation schemes. An important point in this was that the amount of circuitry you need to store numbers depends upon how much precision you want. You need fewer bits of memory to store a number which is guaranteed to be in a small range. The amount of memory that your program uses thus depends upon the number of numbers you need to remember during the computation, and also how precisely you need the number to be represented.

We also saw the notion of a *machine-language program*. A machine language program is a sequence of machine language instructions. A machine-language instruction is a sequence of numbers, which tell the control unit what operation to perform. We noted that the computer designer must design instructions such that every operation that the computer is capable of performing can be expressed as an instruction or as a sequence of instructions. The instructions typically correspond to *primitive* operations, e.g. arithmetic. We also noted that machine language programs typically exist "behind the scenes", i.e. we usually write our programs using a language such as C++, from which an equivalent machine language program is generated automatically by the compiler.

It should be noted that real computers are much more complex than our description in this chapter. However, our description should give you a good enough model of a computer for the purpose of learning to program.

EXERCISES

All the exercises below are meant to be only paper and pencil exercise. No C++ programming is expected.

1. How would you represent a position in a chess game? If you are not familiar with chess, answer this question for any board game you are familiar with.
2. Make sure you are able to convert numbers from decimal to binary and vice versa. You may not be familiar with converting fractions. For this, simply note that a 1 in the ith position after the (binary) point, has place value 2^{-i}. Thus 0.1 in binary is just half, 0.01 is just one fourth. So now you should be able to decide whether the first bit after the point should be one or not, by comparing the fractional part to half. The remaining bits can be decided by extending the idea. You will not need to convert fractions in this book, however, it will be useful to be able to convert intgers routinely. So practice with different examples.
3. How many different numbers are represented in the sign magnitude representation on $n = 3$ bits? Make a table showing what bit pattern represents which number.
4. How many different numbers are represented in the n-bit 2's complement representation on $n = 3$ bits? Make a table showing what bit pattern represents which number.
5. Suppose you want to draw a "+" symbol at the center of a 1024×1024 display. Suppose the display will show a pixel white if you store a 1 at the corresponding memory location. Suppose the "+" is 100 pixels tall and wide, and 2 pixels thick. In which screen memory locations would you store 1s?
6. Consider the problem of computing the fourth power of a number x. One way is to multiply x by itself 3 times. Another way is to first square x, i.e. multiply x by itself, and then square the square, i.e. multiply the computed x^2 by itself. Clearly, the second way is better.
   Adapt the machine-language program of Section 2.9 for cubing a number to compute the fourth power. Write the machine language program for both the methods described above.
   Write a machine language program to compute x^{18} given x. You should be able to do this using just 5 multiplications.
   For now, you will be able to write all this only in machine language, using the single instruction described in Section 2.9. In Chapter 3, you will see how the program can be written using C++.
7. To get a foretaste of how to write programs, imagine that you are to describe to your friend how to perform division over the phone. Yes, you are not allowed to draw any pictures, and the description must be entirely verbal. You may of course talk about using papers with squares and say things like "now read the digit you wrote in the square in the third row and fourth column and multiply it with the digit in the fourth row and fifth column and write the result in ..." Suppose further that your friend is not too imaginative, but very meticulous. In other words, your friend is unwilling to generalize from examples that you might describe. However, your friend will understand if you give a precise description which goes something like "find smallest i such that the number formed by the most significant i digits is larger than the divisor" and so on.

CHAPTER 3

Variables and Data Types

Speaking at a very high level, it could be said that most programs have the following phases: (a) read in some data, (b) perform calculations on the data, (c) print out the results of the calculations or somehow show the results. After reading the data (say as the user types it from the keyboard) it is stored in the memory of the computer. It is then used in the calculations. The results of the calculations, are also stored in memory and then used later in the program for additional calculations or for printing on the screen.

How do we access the memory of a computer from inside a C++ program? This is the main question to be discussed in this chapter. Towards this end, C++ provides the notions of *variables* and *data types*. As we will see, by defining a so-called variable, it is possible to reserve space in memory. You can then place data into that space by reading it from the keyboard, as seen in Chapter 1. But you can also perform calculations and place the results of the calculations into the space you have reserved, using a so called *assignment statement*. We will also study assignment statements.

Using the `repeat` statement and what we learn in this chapter, we will be able to write some interesting programs. Some of these programs are very idiomatic, i.e. similar patterns appear commonly in many programs. We will then see some variations of the assignment statement inspired by these idioms. We will conclude with a discussion of some intricacies related to variables.

3.1 INTRODUCTION

A region of memory allocated for holding a single piece of data (for now a single number), is called a *variable*. C++ allows you to create a variable, i.e. allocate the memory, and give it a name. The name is to be used to refer to the variable in the rest of the program. A variable can be created by writing the following in your program.

```
data-type variable-name;
```

In this, `data-type` must be a data-type selected from the first column of Table 3.1, and `variable-name` a name chosen as per Section 3.1.1. This statement creates a variable of the name `variable-name`, having the specified `data-type`. The `data-type` of a variable determines how much space the variable uses (given in column 3 of Table 3.1), and the type of values expected to be stored in the variable (column 4 of Table 3.1).

You have already seen some examples, e.g. in Section 1.3.1, we wrote

```
int nsides;
```

Table 3.1 *Fundamental data types of C++*

Data type	Possible values (Indicative)	# Bytes Allocated (Indicative)	Use for storing
`signed char`	-128 to 127	1	Characters or small integers.
`unsigned char`	0 to 255		
`short int`	-32768 to 32767	2	Medium size integers.
`unsigned short int`	0 to 65535		
`int`	-2147483648 to 2147483647	4	Standard size integers.
`unsigned int`	0 to 4294967295		
`long int`	-2147483648 to 2147483647	4	Storing longer integers.
`unsigned long int`	0 to 4294967295		
`long long int`	-9223372036854775808 to 9223372036854775807	8	Even longer integers.
`unsigned long long int`	0 to 18446744073709551615		
`bool`	`false` (0) or `true` (1)	1	Logical values.
`float`	Positive or negative. About 7 digits of precision. Magnitude in the range 1.17549×10^{-38} to 3.4028×10^{38}	4	Real numbers.
`double`	Positive or negative. About 15 digits of precision. Magnitude in the range 2.22507×10^{-308} to 1.7977×10^{308}	8	High precision and high range real numbers.
`long double`	Positive or negative. About 18 digits of precision. Magnitude in the range 3.3621×10^{-4932} to 1.18973×10^{4932}	12	High precision and very high range real numbers.

We said then that this would create a variable capable of storing integers. From Table 3.1, you now also know that typically the variable will use 4 bytes of memory, and will store positive and negative numbers. As discussed in Section 2.4.2, such numbers are typically represented in the two's complement representation, and if so the numbers in the range -2147483648 to 2147483647 can be stored.

C++ provides the types `signed char`, `short int`, `long int`, and `long long int` for storing (positive or negative) integers. Variables of these respective types will use amount of memory as given in Table 3.1 and will be able to store values in correspondingly larger or smaller range. In all such cases, very likely the two's complement representation of Section 2.4.2 is used.

If you know that you will only store non-negative integers in a certain variable, you may choose one of the `unsigned` types. For example, you may write:

```
unsigned int telephoneNumber;
```

This will create a variable called `telephoneNumber`, using 4 bytes, and the values will be stored using the binary representation, as discussed in Section 2.4.1. The types `unsigned char, unsigned short, unsigned long` and `unsigned long long` are also used for storing non-negative integers. These will respectively use different amount of memory and allow correspondingly smaller or larger ranges.

The following will create a variable called `temperature` for storing real numbers.

```
double temperature;
```

The created variable will be 8 bytes long. It will typically use the IEEE Floating Point Standard as discussed in Section 2.4.3. The type name `double` is short for "double precision", in comparison to the type `float` which uses 4 bytes and is considered "single precision".

The first 9 types in Table 3.1 are said to be `integral` types, and the last 3, `floating` types.

It should be noted that the size shown for each data type is only indicative. The C++ language standard only requires that the sizes of `char, short, int, long, long long` to be in non-decreasing order. Likewise, the sizes of `float, double, long double` are also expected to be non-decreasing. The exact sizes are may vary from one compiler to another but can be determined as discussed in Section 3.1.6.

The `char` types are most commonly used for storing text, as we will see later. In such uses it is customary to omit the qualifiers `signed` or `unsigned` and write:

```
char firstLetterOfName;
```

This will create a 1 byte variable, of type either `unsigned char` or `signed char`. One of these types will be chosen by the compiler. Note that if you are using `char` to store text, the exact choice does not matter because the ASCII code is uses only the range 0 to 127 which is present in either the signed or the unsigned version. If you use the `char` type to store integers (that happen to lie in a small range) then it is best to specify whether you want the signed or the unsigned type.

The type `bool` is primarily used to store logical values, as will be seen in Section 6.7.

The phrase *value of a variable* is used to refer to the value stored in the variable. So the stored telephone number (after it is stored, and we will say how to do this) will be the value of the variable `telephone_number`.

We finally note that you can define several variables in a single statement if they have the same type, by writing:

```
data-type variable-name1, variable-name2, ... variable-namek;
```

3.1.1 Identifiers

The technical term for a name in C++ is *identifier*. Identifiers can be used for naming variables, but also other entities as we will see later.

An identifier can consist of letters, digits and the underscore character "_". Identifiers cannot start with a digit, hence you cannot have an identifier such as `3rdcousin`. It is also not considered good practice to use identifiers starting with an underscore for naming ordinary variables. Finally, some

words are reserved by C++ for its own use, and these cannot be used as variable names. For example, `int` is a reserved word; it is not allowed to be used as a variable name because it will be confusing. The complete list of reserved words is given in Appendix B.

It is customary to name a variable to indicate the intended purpose of the variable. For example, if we want to store a velocity in a variable, then we should give the name `velocity` to the variable.

An important point is that case is important in names; so `mathmarks` is considered to be a different name from `MathMarks`. Notice that the latter is easier to read. This way of forming names, in which several words are strung together, and in which the first letter of each word is capitalized, is said to be utilizing camel case, or CamelCase. As you might guess, the capital letters resemble the humps on the back of a camel. There are two kinds of CamelCase: UpperCamelCase in which the first letters of all the words are capitalized, and lowerCamelCase, in which the first letters of all but the first word are capitalized. For ordinary variables, it is more customary to use lowerCamelCase; thus it is suggested that you use `mathMarks` rather than `MathMarks`.

If a variable is important in your program, you should give it a descriptive name, which expresses its use. It is usually best to use complete words, unabbreviated. Thus if you have a variable which contains the temperature, it is better to give it the name `temperature` rather than `t`, or `temp` or `tmprtre`. Sometimes the description that you want to associate with a variable name is very long. Or there is a clarification that the reader should be be aware of. In such cases, it is good to add a comment explaining what you want immediately following the definition, e.g.

```
double temperature;              // in degrees centigrade.
```

3.1.2 Literals and Variable Initialization

It is possible to optionally include an initial value along with the definition. So we may write

```
int p=10239, q;
```

This statement defines 2 variables, of which the first one, `p`, is initialized to 10239. No initial value is specified for `q`, which means that some unknown value will be present in it. The number 10239 as it appears in the code above is said to constitute an integer *literal*, i.e. it is to be interpreted literally as given. Any integer number with or without a sign constitutes an integer literal. The words `false` and `true` are literals which stand for the values 0 and 1. So for `bool` variables, it is recommended that you write initializations using these, e.g.

```
bool penIsDown = true;
```

rather than writing `bool penIsDown = 1;` which would mean the same thing but would be less suggestive. For convenience in dealing with `char` data, any character enclosed in a pair of single quotes is an integer literal that represents the ASCII value of the enclosed character. Thus you may write

```
char letter_a = 'a';
```

This would store the code, 97, for the letter 'a' in the variable `letter_a`. You could also have written `char letter_a = 97;` but writing `'a'` is preferred, because it is easier to understand. In general, we may write a character between a pair of single quotes, and that would denote the ASCII value of the character. Characters such as the newline (produced when you press the "enter" key), or the tab, can be denoted by special notation, respectively as '\n' and '\t'. Note that literals such as '\n' and 'a' really represent an integer value. So we can in fact write

```
int q = 'a';
```

This would cause 97 to be stored in the `int` variable `q`.

To initialize floating variables, we need a way to specify real number literals. We can specify real number literals either by writing them out as decimal fractions, or using an analogue of "scientific notation". We simply write an `E` or `e` between the significand and the exponent, without leaving any spaces. Thus, we would write Avogadro's number[1], 6.022×10^{23}, as `6.022E23`. The significand as well as the exponent could be specified with a minus sign, if needed, of course. For example the mass of an electron, $9.10938188 \times 10^{-31}$ kg, would be written as `9.10938188E-31`. Thus we may write:

```
float w, y=1.5, avogadro = 6.022E23, eMass = 9.10938188E-31;
```

This statement defines 4 variables, the second, third and fourth are respectively initialized to 1.5, 6.022×10^{23} and $9.10938188 \times 10^{-31}$. The variable `w` is not initialized.

Literals also have a type associated with them. An integer literal like 35 is considered to be of type `int`, and a floating literal like 100.0 is by default considered to be of type `double`. You can specify literals of specific types by attaching the suffixes L,F,U which respectively stand for long, float, unsigned. Thus, if you write `100LU`, it will be interpreted as a literal of type `long unsigned`, having the value 100.

3.1.3 The `const` Keyword

Sometimes we wish to define identifiers whose value we do not wish to change. For example, we might be needing Avogadro's number in our program, and it will likely be convenient to refer to it using the name `Avogadro` rather than typing the value everytime. In C++ you can use the keyword `const` before the type to indicate such named constants. Thus, you might write

```
const float Avogardro = 6.022E23;
```

Once a name is declared `const`, you cannot change it later. The compiler will complain if do attempt to change it.

3.1.4 Reading Data into a Variable

To read a value into a variable `pqr` we write

```
cin >> pqr;
```

Simply put: when this statement is executed, the computer will wait for us to type a value consistent with the type of `pqr`. That value will then be placed in `pqr`.

The exact execution process for the statement is a bit complicated. First, the statement ignores any *whitespace* characters that you may type before you type in the value consistent with the type of `pqr`. The term whitespace is used to collectively refer to several characters including the space character (' '), the tab character ('\t'), and the newline character ('\n'). In addition, the vertical tab ('\v'), the formfeed character ('\f') and the carriage return ('\r') are also considered whitespace. These three characters are now only of historical interest.

The first non-whitespace character you type is considered to be the start of the value you wish to give for `pqr`. You may type several non whitespace characters as value if appropriate. After typing the desired value you must type a whitespace character (often newline) to signify that you have finished

[1] The number of molecules in a mole of any substance, e.g. number of carbon atoms in 12 gm of carbon.

typing the value that you wanted. Let us consider an example. Suppose `pqr` has type `int`, then if you execute the above statement, and type

```
123 56
```

the spaces that you type at the beginning will be ignored, the value 123 will be stored into `pqr`. This is because the space following 123 will serve as a delimiter. The 56 will be used for a subsequent read statement, if any. Note further that the value you type will not be received by your program unless you type a newline after typing the value. Thus to place `123` into `pqr` in response to the statement above, you must type a newline either immediately following 123 or following 56.

If `pqr` was of any of the floating types, then a literal of that type would be expected. Thus we could have typed in `6.022e23` or 1.5. If `pqr` was of type `bool` you may only type 0 or 1.

❑ *Reading into a* `char` *Variable*

You may not perhaps expect what happens when you execute

```
char xyz;
cin >> xyz;
```

In this case the initial whitespaces that you type if any will be ignored, as discussed above. Any non-whitespace value is considered appropriate for the type `char`, so the first such value will be accepted. The ASCII value of the first non-whitespace character that you type will be placed into `xyz`. Note that if you type 1, then `xyz` will become 49. This is because the ASCII value of the character '1' is 49. If you type the letter a, then `xyz` would get the value 97.

❑ *Reading Several Values*

If you wish to read values into several variables, you can express it in a single statement.

```
cin >> pqr >> xyz;
```

This is equivalent to writing `cin >> pqr; cin >> xyz;`.

3.1.5 Printing

If you print a variable `rst` of type `bool, short, int` or `long`, writing

```
cout << rst << endl;
```

its value will be printed. A minus sign will be printed if the value is negative. The final `endl` will cause a newline to follow.

If you print a floating type variable, then C++ will print it in what it considers to be the best looking form: as a decimal fraction or in the scientific format.

❑ *Printing a* `char` *Variable*

Consider the following code.

```
char xyz=97;
cout << xyz << endl;
```

This will cause that character whose ASCII value is in `xyz` to be printed. Thus, in this case, the letter a will be printed. Following that a newline will be printed, because of the `endl` at the end of the statement.

❑ *Printing Several Values*

The two previous statements above can be combined into a single statement if you wish.

```
cout << rst << endl << xyz << endl;
```

Occasionally, you may wish to control exactly how the printing happens, e.g. how many bits are shown after the decimal point. This is discussed in Appendix D.

3.1.6 Exact Representational Parameters

Table 3.1 mentions the indicative sizes of the different data types. You can find the exact number of bytes used by your compiler by using the `sizeof` command in your program:

```
cout << sizeof(int) << endl;
```

Or `sizeof(double)` and so on as you wish. You can also write `sizeof(variable-name)` to get the number of bytes used for the variable `variable-name`.

You can also determine the largest or smallest (magnitude) representable numbers in the different types. Say for `float`, the expression `numeric_limits<float>::max()` gives the value of the largest floating point number that can be represented. Please do not worry about the complicated syntax of this expression. By using other types instead of `float` or by using `min` instead of `max`, you can get the minimum/maximum values for all types. In order to use this facility, you need to put the following line at the top of your file (before or after other `#include` statements):

```
#include <limits>
```

We will see the exact action of this line later.

3.2 ARITHMETIC AND ASSIGNMENT

We can perform arithmetic on the values stored in variables in a very intuitive manner, almost like we write algebraic expressions. The values resulting from evaluating an arithmetic expression can be stored into a variable by using an assignment statement.

The notion of expressions is similar to that in Algebra. If you have an algebraic expression $x \cdot y + p \cdot q$, its value is obtained by considering the values of the variables x, y, p, q, and performing the operations as per the usual precedence rules. In a similar manner you can write expressions involving C++ variables, and the value of the expression is obtained by similarly considering the values of the variables and performing operations on them, with similar rules of *operator precedence*. One difference is that often in Algebra the multiplication operator is implicit, i.e. xy means x multiplied by y. In a C++ expression, we need to explicitly write the multiplication operator, which is $\star$. All the arithmetic operators `+, -, *, /` are allowed. Multiplication and division have equal precedence, which is higher than that of addition and subtraction which have the same precedence. Some additional operators are also allowed, as will be discussed later. Among operations of the same precedence, most commonly, the one on the left is performed first, e.g. 5 − 3 + 9 will mean 11. However, for some

operators, i.e. the so called *right-associative* operators, the operation on the right will get done first (Section 3.2.6). Of course, we can use brackets to enforce the order we want, e.g. write 5 − (3 + 9) if we want this expression to evaluate to −7. If we had C++ variables `x,y,p,q`, then the expression corresponding to the algebraic expression above would have to be written as `x*y+p*q`. Note that when you use a variable in an expression, it is your responsibility to ensure that the variable has been assigned a value earlier.

An expression causes a sequence of arithmetic operations to be performed, and a value to be computed. However, the computed value is lost unless we do something with it. One possibility is to store the computed value in some variable. This can be done using an assignment statement. The general form of an assignment is

```
variable = expression;
```

where `variable` is the name of a variable, and `expression` is an expression as described above. Here is an example.

```
int x=2,y=3,p=4,q=5,r;
r = x*y + p*q;
```

This will cause `r` to get the value of the specified expression. Using the values given for the other variables, the expression is simply 2 * 3 + 4 * 5, i.e. 26. Thus `r` will get the value 26.

We could also print out the value of the expression by writing

```
cout << x*y+p*q << endl;
```

Note that when you use a variable in an expression, you must have assigned it a value already, say by initializing it at the time of defining it, or by reading a value into it from the keyboard, or in a previous assignment statement. If this is not done, the variable will still contain some value, only you don't know what value. If an unknown value is used in a computation, the result will of course be unpredictable in general.

Note that the operator = is used somewhat differently in C++ than in mathematics. In mathematics a statement `r = x*y + p*q;` asserts that the left-hand side and right-hand side are equal. In C++ however, it is a command to evaluate the expression on the right and put the resulting value into the variable named on the left.

Note also that we cannot write `x*y + p*q = r;` because we require the left hand side to be a variable, into which the value of the expression on the right hand side must be stored.

The rule described above makes it perfectly natural to write a statement such as

```
p = p + 1;
```

This is meaningless in mathematics; in C++, however, it just says: evaluate the expression on the right hand side and put the resulting value into the variable named on the left. Assuming `p` has the value 4 as in the code fragment given earlier, its value is 4, the statement would cause the addition 4 + 1 = 5 to be performed. The result, 5, would be put in `p`.

3.2.1 Integer Division and the Modulo Operator `%`

In C++, when one integer value is divided by another, the result is defined to also be the largest integer no larger than the quotient. Thus, if you write

```
int m=100, n=7, p, q;
p = m/n;
q = 35/200;
```

the variables `p` and `q` would respectively get the values 14 and 0. In other words, we only get the integer part of the quotient.

If you wish to get the remainder resulting when one integer divides another you use the `%` operator. Thus the expression `m % n` evaluates to the remainder of `m` when divided by `n`, where `m,n` must have an integral type. The operator `%` has the same precedence as `*` and `/`.

Here is a code fragment that reads in a duration given in seconds and prints out the equivalent duration in hours, minutes, and seconds.

```
cout <<"Give the duration in seconds: ";
int duration; cin >> duration;
int hours, minutes, seconds;
hours = duration/3600;
minutes = (duration - hours*3600)/60;
seconds = duration % 60;
cout <<"Hours: "<< hours <<", Minutes: "
     << minutes <<", Seconds: "<< seconds << endl;
```

If you run this code, and type 5000 when asked, you would get the following output as expected:

```
Hours: 1, Minutes: 23, Seconds: 20
```

3.2.2 Subtleties

The assignment statement is somewhat tricky. The first point concerns the floating point representations. Both `float` and `double` are imprecise representations, where the significand is correct only to a fixed number of bits. So if an arithmetic operation affects less significant bits, then the operation will have no effect. As an example, consider the following code.

```
float w, y=1.5, avogadro=6.022E23 ;
w = avogadro + y;
```

What is the value of `w`? Suppose for a moment that we precisely calculate the sum `avogadro + y`. The sum will be

$$602200000000000000000001.5$$

We will have a problem when we try to store this into a `float` type variable. This is because a `float` type variable can only stores significands of 24 bits, or about 7 digits. So in order to store, we would treat everything beyond the most significant 7 digits as 0. If so we would get

$$602200000000000000000000$$

This loss of digits is called *round-off error*. After the round off, this can now fit in a `float`, because it can be written exactly as `6.022E23`. Net effect of the addition: nothing! The variable `w` gets the value `avogadro` even though you assigned it the value `avogadro + 1.5`. This example shows the inherent problem in adding a very small `float` value to a very large `float` value.

Some subtleties arise when we perform an arithmetic operation in which the operands have different types, or even simply if you store one type of number into a variable of another type. C++ allows such operations, and could be said to perform such actions reasonably well. However, it is worth knowing what exactly happens.

Suppose we assign an `int` expression to a `float` variable, C++ will first convert the expression into the floating point format. An `int` variable will have 31 bits of precision excluding the sign, whereas a `float` variable only has 24 bits or so. So essentially some precision could be lost. There could be loss of precision also if we assign a `float` expression to an `int` variable. Consider

```
float y = 6.6;
int x = y;
```

The value 6.6 is not integral, so C++ tries to do the best it can: it keeps the integer part. At the end, `x` will equal 6. Basically, when a floating value is to be stored into an integer, C++ uses truncation, i.e. the fractional part is dropped. You might want the assigned value to be the closest integer. This you can obtain for yourself by adding 0.5 before the assignment. Thus, if you write `x=y+0.5;`, then `x` would become 7, the integer closest to `y`. Note that some precision could be lost when you store a value from a `double` (53 bits of precision) into a float (24 bits of precision). Overflow is also possible, as discussed later.

When we perform an arithmetic operation on operands of the same type the result is also computed to be of the same type. If your program asks to perform arithmetic operations on operands of different types, then the operands are first converted by C++ so that they have the same type. The rules for this are fairly natural. C++ always converts less expressive types to more expressive ones, where unsigned integral types are deemed less expressive than signed integral types, which in turn are deemed less expressive than the floating types. If the two types differ in size, then the smaller is converted to have a larger size. As an example, suppose we have an arithmetic expression `var1 op var2`, where var1 is `int` and `var2` is `float`. Then `var1` will be converted to `float`, and the result will also be `float`. If `var1, var2` are `long, int,` then `var2` will be converted to `long`. If the operands are of type `float, long long` then both will be converted to `double`, and so on. After the expression is evaluated, it may either itself form an operand in a bigger expression, or it might have to be stored into a variable. In both cases, there may have to be a further type conversion.

It is important to be careful with division.

```
int x=100, w;
float y,z;
y = 360/x;
z = 360.0/x;
w = 360.0/x;
```

As per the rules stated, `360/x` will be evaluated to have an integer value since both operands are integer. Thus the exact quotient 3.6 will be truncated to give 3. This value will be stored (after conversion to the floating point format) into `y`. In the next statement, `360.0`, the first operand is `double`, hence the result will be evaluated as a `double`, i.e. 3.6. This value will be stored in `z`. In the final statement, the value of the expression will indeed be 3.6, however because `w` is of type `int`, there will have to be a type conversion, and as a result the value stored in `w` will be just 3.

Note finally that if the dividend and the divisor are of integral types, and the divisor is 0, then an error will be reported when such an operation happens during execution, and the program will stop with a message. Something different happens for floating types, as discussed in Section 3.2.4.

3.2.3 Overflow

For each numerical data type, we have mentioned a certain largest possible and smallest possible value that can be represented. While performing calculations, the results can go outside this allowed range. In this case, what exactly happens is handled differently for different types.

For the `unsigned` data types, the rule is that arithmetic is performed modulo 2^n, where n is the number of bits used. So for example if you add up two `short` numbers, both 65535, then the result will be $(65535 + 65535) \bmod 65536 = 65534$, where you may note that $2^{16} = 65536$.

For signed integer types, the language does not specify what must happen. In other words, you as a programmer must be careful to ensure that the numbers stay within range.

3.2.4 Infinity and Not a Number

Most C++ compilers support the IEEE floating point standard. With such compilers, something quite interesting happens if the result of a floating type computation becomes too large to represent, e.g. if you try to compute the square of Avogadro's number and try to store it into a `float` variable. In such cases, a special bit pattern gets stored in the variable. This bit pattern behaves like infinity for all subsequent computation. By this, we mean that anything added to infinity remains infinity, and so on. If you try to print out this pattern, quite likely `inf` would get printed. Thus, you at least get some indication that some overflow occurred during computation. You also get the result `inf` when you divide a positive floating value by 0. Likewise, you get `-inf` when you divide a negative floating number by 0.

If the dividend and the divisor are both zeros, represented as floating point numbers, then you get another special bit pattern which will likely be printed as `nan`. This pattern is meant to represent the result of an undefined operation, `nan` is an abbreviation for "not a number". If you happen to use a variable or an expression of value `nan` in any operation, the result will also be `nan`. Note that taking the square root of a negative number also produces `nan`.

We will see later that it is actually useful to use infinities in our computations. In your C++ programs you can refer to ∞ using the name `HUGE_VAL`. Thus, you may write

```
double x = HUGE_VAL;
```

3.2.5 Explicit Type Conversion

It is possible to convert an expression `exp` of numerical type `T1` to an expression of type `T2` by writing either

```
T2(exp)
```

or

```
(T2) exp
```

This latter form is a legacy from the C language. The type-conversion rules as described earlier apply, e.g. `int(6.4)` would evaluate to the integer value 6.

3.2.6 Assignment Expression

It turns out that C++ allows you to write the following code.

```
int x,y,z;
x = y = z = 1;
```

This will end up assigning 1 to all the variables. This has a systematic explanation as follows.

Any assignment, say `z = 1`, is also an expression in C++. Not only is the assignment made, but the expression stands for the value that got assigned. Further, the *associativity* of = is right-to-left, i.e. given an expression `x = y = z = 1`, the rightmost assignment operator is evaluated first. This is different from the other operators you have seen so far, such as the arithmetic operators, in which the evaluation order is left to right. Thus, the our statement `x = y = z = 1;` is really to be read as

```
x = (y = (z = 1));
```

Now the expression inside the innermost parentheses, `z = 1` is required to be evaluated first. This not only puts the value 1 into `z`, but itself evaluates to 1. Now the statement effectively becomes

```
x = (y = 1);
```

The execution continues by setting `y` to 1, and then `x` to 1.

3.3 EXAMPLES

We consider some simple examples of using the data-types and assignment statements. These do not include the `bool` type which is considered in Section 6.7.

Here is a program that reads in the temperature in Centigrade and prints out the equivalent temperature in Fahrenheit.

```
main_program{
  double centigrade, fahrenheit;

  cout << "Give temperature in Centigrade: ";
  cin >> centigrade;

  fahrenheit = 32.0 + centigrade * 9.0/5.0;
  cout << "Temperature in Fahrenheit: " << fahrenheit << endl;
}
```

Note that the operator + is executed last because it has lower precedence than `*` and `/`. The operator `*` executes before `/` because it appears to the left. Note we could have written 9 instead of 9.0. This is because that while multiplying `centigrade`, it would get converted to a `double` value anyway, since `centigrade` is `double`. Similarly we could have written 5 and 32 instead of 5.0 and 32.0. But what we have written is preferable because it makes it very clear that we are engaging in floating point arithmetic.

In the next program, you are expected to type in any lowercase letter, and it prints out the same letter in the uppercase.

```
main_program{
  char small, capital;

  cout << "Type in any lowercase letter: ";
  cin >> small;

  capital = small + 'A' - 'a';

  cout << capital << endl;
}
```

When the statement `cin >> small;` executes, the ASCII value of the letter typed in by the user is placed in `small`. Suppose as an example that the user typed in the letter q. Then its ASCII value, `'q'` is placed in `small`. This value happens to be 113. To understand the next statement, we need to note an important property of the ASCII codes.

The lowercase letters a-z have consecutive ASCII codes. The upper case letters A-Z also have consecutive ASCII codes. From this, it follows that for all letters, the difference between the ASCII code of the uppercase version and the lowercase version is the same. Further, because 'A' and 'a' denote the integers representing the ASCII codes of the respective letters, 'A' − 'a' merely gives the numerical difference between the ASCII codes of upper case and lower case of the letter a. But this difference is the same for all letters. Hence, given the ASCII code value for any lowercase letter, we can add to it 'A' − 'a', and this will give us the ASCII code of the corresponding uppercase letter. So this value gets placed in `capital`, which when printed out displays the actual uppercase letter.

To complete the example, note that the ASCII code of 'A' is 65. Thus 'A' − 'a' is −32. Since `small` contains 113, capital would get 113 − 32, i.e. 81. This is indeed the ASCII code of Q as required.

Note that the digits '0', '1', '2', . . ., '9' also have consecutive ASCII codes.

3.4 ASSIGNMENT WITH `repeat`

What do you think happens on executing the following piece of code?

```
main_program{
  turtleSim();
  int i = 1;
  repeat(10){
    forward(i*10); right(90);
    forward(i*10); right(90);
    i = i + 1;
  }
  wait(5);
}
```

Imagine that you are the computer and execute the code one statement at a time. Write down the values of different variables as you go along, and draw the lines traced by the turtle as it moves. You will probably be able to figure out by executing 2-3 iterations. It is strongly recommended that you do this before reading the explanation given next.

In the first iteration of the `repeat`, `i` will have the value 1, and this value will increase by 1 at the end of each iteration. The turtle goes forward `10*i`, i.e. a larger distance in each iteration. As you will see, the turtle will trace a "spiral" made of straight line segments.

We next see another common but important interaction of the assignment statement and the `repeat` statement. Consider the following problem. We want to read some numbers, from the keyboard, and print their average. For this, we need to first find their sum. This can be done as follows.

```
main_program{
  int count;
  cout << "How many numbers: ";
  cin >> count;

  float num,sum=0;
  repeat(count){
    cout << "Give the next number: ";
    cin >> num;
    sum = sum + num;
  }

  cout << "Average is: ";
  cout << sum/count;
  cout << endl;
}
```

The statement `sum = sum + num;` is executed in each iteration, and before it is executed, the next number has been read into `num`. Thus, in each iteration the number read is added into `sum`. Thus, in the end `sum` will indeed contain the sum of all the numbers given by the user.

3.4.1 Programming Idioms

There are two important programming idioms used in the programs of the previous section.

The first idiom is what we might call the *sequence-generation idiom*. Note the value of the variable i in the first program. It started off as 1, and then became 2, then 3, and so on. As you can see, by changing the starting value for `i` and adding a different number to `i` inside the loop instead of 1, we could make `i` take the values of any arithmetic sequence (Exercise 7). By changing the operator to $\star$ instead of +, we could make the values form a geometric sequence if we wished.

The second idiom is what we might call the *accumulation idiom*. This was seen in the second program. The variable `sum` was initialized to zero, and then the number read in each iteration was added to the variable `sum`. The variable `sum` was thus used to *accumulate* the values read in each iteration. Stating this differently, suppose the number of numbers read is n, and suppose the values read were $v_1, \ldots, v_n$. Then after the execution of the loop in the second program the variable `sum` has the value:

$$0 + v_1 + v_2 + \cdots + v_n$$

Here, we have written $0+$ explicitly to emphasize that the value calculated actually also depends on the value to which `sum` was initialized, and that happened to be zero, but it is a choice we made.

You might wonder whether this idea only works for addition or might work for other operators as well. For example, C++ has the command `max`, where `max(a,b)` gives the maximum of the values of the expressions `a,b`. Will using `max` help us compute the value of the maximum of the values read? In other words, what would happen if we defined a variable `maximum` and wrote

```
maximum = max(maximum, num);
```

instead of `sum = sum + num;`? For simplicity, assuming $n = 4$ and also assuming that `maximum` is initialized to 0 just as `sum` was, the value taken by `maximum` at the end of the repeat will be:

$$\max(\max(\max(\max(0, v_1), v_2), v_3), v_4)$$

Will this return the maximum of v_1, v_2, v_3, v_4? As you can see, this will happen only if at least one of the numbers is positive. If all numbers are negative, then this will return 0, which is not the maximum. Before we abandon this approach as useless, note that we actually have a choice in deciding how to initialize `maximum`. Clearly, we should initialize it to as small a number as possible, so that the values v_i cannot be even smaller. We know from Section 3.1.6 that it suffices to choose `-numeric_limits<float>::max()`. Thus our initialization becomes:

```
maximum = - numeric_limits<float>::max();
```

which we put in place of the statement `sum=0;` in the program.

There is another way to do this also, which you might find simpler. We could merely read the first value of `num`, and assign `maximum` to that. Thus the program just to calculate the maximum of a sequence of numbers will be as follows. Note that we now repeat only `count-1` times, because we read one number earlier.

```
main_program{
  int count;
  cout << "How many numbers: ";
  cin >> count;

  float num,maximum;

  cout << "Give the next number: ";
  cin >> maximum;

  repeat(count-1){
    cout << "Give the next number: ";
    cin >> num;
    maximum = max(maximum,num);
  }
  cout << "Maximum is: " << maximum << endl;
}
```

This program does not behave identically to the program sketched earlier, i.e. obtained by initializing `maximum` to `- numeric_limits<float>::max()`. The exercises ask you to say when the programs might differ, and which one you prefer under what circumstances.

3.4.2 Combining Sequence Generation and Accumulation

Often we need to combine the sequence generation and accumulation idioms.

Suppose we want to compute n factorial, written as $n!$, which is just short hand for the product $n \times (n-1) \times (n-2) \times \cdots \times 2 \times 1$. How can we do this?

The key point to note is that we merely need to take the product of the sequence of numbers $1, 2, \ldots, n-1, n$, and this is a sequence that we can generate. But if we can generate a sequence, then we can easily take its product, i.e. accumulate it using the multiplication operator.

```
main_program{
  int n, fac=1, i=1;
  cin >> n;

  repeat(n){
    fac = fac * i;        // sequence accumulation
    i = i + 1;            // sequence generation
  }
  cout << fac << endl;
}
```

In the above program, if you ignore the statement `fac = fac * i;`, then we merely have our sequence generation program, with the sequence 1 to `n` being generated in the values taken by the variable `i`. However, the value generated in each iteration is being multiplied into the variable `fac` which was initialized to 1. Hence in the end the variable `fac` contains the product of the numbers from 1 to `n`, i.e. `n!` as we wanted. Note that the program also works for `n=0`, since 0! is defined to be 1.

The idioms of accumulation and sequence generation are very useful, and very, very commonly used. They are used so commonly that they will become second nature soon. We see a more involved example in Chapter 4.

3.5 SOME OPERATORS INSPIRED BY THE IDIOMS

Because sequence generation and accumulation occur commonly in code, C++ includes operators that can be used to express these idioms more succinctly.

3.5.1 Increment and Decrement Operators

A key statement in the sequence generation idiom is `i=i+1;`. This tends to occur quite frequently in C++ programs. So a short form has been provided. In general you may write

```
C++;
```

which merely means `C = C + 1;`, where `C` is any variable. This usage is very useful.

For completeness, we describe some additional, possibly confusing, feature of the `++` operator. Turns out that for a variable `C`, `C++` is also an expression. It stands for the value that `C` had before 1 was added to it. Thus, if you wrote

```
int x = 2, y;
y = x++;
```

after execution, `y` would be 2 and `x` would be 3. We recommend that you avoid a statement such as `y = x++;` and instead write it as the less confusing `y = x; x++;`. It is worth noting that in the

modern era programming is often done by teams, and so your code will be read by others. So it is good to write in a manner that is easy to understand quickly.

The operator ++ written after a variable is said to be the (unary) *post-increment* operator. You may also write `++C`, which is the unary *pre-increment* operator operating on the variable `C`. This also causes `C` to increase by 1. `++C` also has a value as an expression: except the value is the new value of `C`. Thus, if you wrote

```
int x = 2, y;
y = ++x;
```

both `x,y` would get the value 3. Again, this will usually be better written as `++x; y = x;` (or for that matter as `x++; y = x;`) because it is less confusing.

Likewise, `C--`; means `C = C - 1;`. This is also a very useful operator, and is called the post decrement operator. As an expression, `C--` has the value that `C` had before the decrementation. Analogously, you have the pre-decrement operator with all similar properties. Again, it is recommended that you use the expression forms sparingly.

3.5.2 Compound Assignment Operators

The accumulation idiom commonly needs the statement `vname = vname + expr;`, where `vname` is a variable name, and `expr` an expression. This can be written in short as

```
vname += expr;
```

The phrase `vname += expr` is also an expression and has as its value the value that got assigned. Analogously, C++ has operators `*=`, and `-=`, `/=`. These operators are collectively called the compound assignment operators.

The expression forms of the operator += and others are also defined in the natural manner: the value of the expression is the value that got assigned. It is recommended that you use these expression forms sparingly.

3.6 BLOCKS AND VARIABLE DEFINITIONS

It turns out that most C++ programmers would write the average computation program from Section 3.4 slightly differently, as follows.

```
main_program{
  int count;
  cout << "How many numbers: ";
  cin >> count;

  float sum=0;
  repeat(count){
    cout << "Give the next number: ";
    float num;                          // defined inside the loop
    cin >> num;
    sum = sum + num;
  }
```

```
  cout << "Average is: ";
  cout << sum/count;
  cout << endl;
}
```

As you can see, the only difference is that the variable `num` is defined inside the loop rather than outside. We first explain how the variable definition is executed in the new program. As you might guess, the variable indeed gets created when control reaches the definition statement. From the time of creation, the variable is available to the program, *until the time the control reaches the end of the loop body in the current iteration*. In other words, the variable is *destroyed* when the control reaches the end of the body! Thus, in each iteration of the loop, the variable is created and destroyed. Of course, *destroying* a variable is only notional, the computer merely assumes that the memory that was given is now available for other use. It should also be noted that the variable cannot be used outside the repeat loop, or before its definition inside the loop.

Experienced programmers prefer to write the average computation code in the new style, because in this the definition of `num` is placed close to its use. Placing definitions close to the use makes it easier to read the program, especially if it has many variables and the loop bodies are large.

Next we will state the general rules for all this. First, we need the notion of a *block*.

3.6.1 Block

The region of the program from an opening brace, `{`, to the corresponding closing brace, `}`, is called a *block*. Thus, the entire program forms a block, and the body of a repeat also forms a block, which is contained inside the block consisting of the entire program. If there is a `repeat` inside a `repeat`, then the block corresponding to the body of the former is contained inside the block associated with the latter. As you can see, two blocks must either be completely disjoint, or one of them must be completely contained in the other. It is also useful to define the *parent block* of a variable definition: it is the innnermost block in which the variable is defined.

3.6.2 General Principle 1: Scope

Now, we can restate more formally what we stated earlier. When control reaches a variable definition, the corresponding variable is created. The variable is destroyed when the control leaves the parent block of the definition. The variable is potentially available for use in the region of the program starting at the point of its definition, and going to the end of its parent block. This region of the program is called the *scope* of the definition.

We have already discussed how this principle applies to the variable `num` of the program given above.

The principle also applies to the variable `sum` in the program. Its parent block is the main program itself, and indeed, the entire portion of the program from the point of its definition to the end of the program can refer to the variable `sum`.

3.6.3 General Principle 2: Shadowing

The principles in giving names to variables and using the names, are somewhat similar to the way in which we give names to human beings.

Let us first discuss how we name human beings. Ideally, you might think that we should insist that all human beings be given different names. But of course, this does not happen. It is perfectly possible

that there exist two families in Mumbai both of which name their son Raju. In that case whenever a reference is made to "Raju" in either family, it is deemed to refer to the son in that family. There is no confusion. Notice, however, that usually the same name is not given to two children in the same family.

As another example, consider the name *Manmohan*. In most families in India, the name would be considered referring to the Prime Minister of India.[2] Suppose now that a certain family decides to name their son "Manmohan". In this family, after the birth of the son, if anyone speaks of Manmohan, it would probably be considered as referring to the son. You could say that the son "overshadows" the Prime Minister in this family.

Variable naming in C++ is almost as flexible as naming of human beings, including the idea of shadowing. The analogue of the family is a block of the program.

In a C++ program, it is possible to use the same name in several variable definitions. However, it is necessary that the definitions have different parent blocks. Even if there are many variable definitions for the same name, the rules for creating and destroying variables remain the same. A variable is created when the definition is encountered during execution, and is destroyed when its parent block is exited. Or alternately, a variable is created when control enters the scope of the definition and is destroyed when control leaves the scope of the definition. Suppose now that the control has entered the scope of a certain definition that creates a variable `Q`. As the execution proceeds, but before the variable is destroyed, suppose we have another definition, also of variable `Q`. Now a second variable named `Q` will be created and while control is inside the scope of the second definition, the second variable will shadow the first variable. In other words, inside the scope of the second definition the name `Q` will not refer to the first variable. It will instead refer to the second variable – unless that of course is shadowed by a third definition of `Q`.

Here are some examples.

```
main_program{
  int sum=0;
  repeat(5){
    int num;                  // statement 1
    cin >> num;
    sum += num;
  }
  cout << sum << endl;
  int prod=1;
  repeat(5){
    int num;                  // statement 2
    cin >> num;
    prod *= num;
  }
  cout << prod << endl;
}
```

In this case, the references to `num` in the first loop are in the scope of the definition in statement 1 (and of no other definition), and hence refer to the variable created in statement 1. Similarly, the references to `num` in the second loop are in the scope of the definition in statement 2 (and of no other definition),

[2] At the time of writing this book.

and hence refer to the variable created in statement 2. This is what you would intuitively expect, and indeed the program will compute the sum of the first 5 numbers that it reads, and the product of the next 5.

Here is an example of a program in which there is shadowing.

```
main_program{
  int p=10;              // statement 3
  repeat(3){
    cout << p << endl;   // statement 4
    int p=5;             // statement 5
    cout << p << endl;   // statement 6
  }
  cout << p << endl;     // statement 7
}
```

In this program, the occurence of `p` in statement 6 is in the scope of the definitions in statements 3 and 5, with the latter shadowing the former. Thus, the name `p` in statement 6 refers to the variable created in statement 5. However, note that the statements 4 and 7 are only in the scope of the definition in statement 3. Thus, the name `p` in this statements refers to the definition in statement 3.

Thus, when control arrives at statement 3, a variable `p` is created. When control arrives at statement 4, the value of this `p`, 10, is printed. When control arrives at statement 5, a new variable, also named `p`, will be created, and will start shadowing the definition of statement 3. At the end of the loop, the variable created in statement 5 will be destroyed. Thus when control reached statement 7, the variable created in statement 5 will have been destroyed, and the statement is in the scope only of the definition in statement 3. Thus, the reference to `p` in statement 7 will be considered to be to the variable `p` defined in statement 3. Thus statement 7 will cause 10 to be printed. Thus, the entire code when executed will cause the sequence of numbers 10, 5, 10, 5, 10, 5, 10 to be printed.

3.7 CONCLUDING REMARKS

The initial part of most programs consist of statements which reserve memory in which to store data. Such statements are called *variable definition statements*. A definition reserves the space and also gives it a name. The reserved space, together with its name, is said to constitute a *variable*, and the data stored in the variable is said to be the value of the variable. Of course, what is stored in memory is always a sequence of bits. The value represented by the bit sequence depends upon how we interpret the bits, which is specified by the *type* of the variable. As discussed in Chapter 2, it is possible that the same pattern of 32 bits might mean one value for a variable of type `unsigned int`, another for a variable of type (signed) `int`, and yet another for a variable of type `float`.

When we define a variable, it is always for some specific purpose. So it is strongly recommended that the name chosen for the variable reflect that purpose. Also, along with the definition, it is useful to write additional comments which explain its purpose in more detail if necessary.

When we mention the name of a variable in a program, it almost always means the *value* stored in the variable, except when the name appears on the left side of an assignment statement, when it means the memory associated with the variable. Perhaps this observation is useful to prevent being confused by statements such as `p = p + 1;` which are incorrect in mathematics but which are meaningful in computer programs. We also noted that the assignment statement is somewhat subtle, because of issues such as rounding, and converting between different types of numbers.

The assignment statement also plays a central role in two important programming idioms: sequence generation, and accumulation. We saw a number of operators which can be considered to have been inspired by these idioms.

We noted that it is convenient if a variable is defined close to the point in the program where it is used. This led us to notion of the scope of a variable, i.e. the region of the program where the variable can be referred to, and also the notion of shadowing.

EXERCISES

1. What is the value of `x` after the following statements are executed? (a) `x=22/7;` (b) `x=22.0/7;` (c) `x=6.022E23 + 1 - 6.022E23` (d) `x=6.022E23 - 6.022E23 + 1` (e) `x=6.022E23 * 6.022E23`. Answer for three cases, when `x` is defined to be of type `int`, `float`, `double`. Put these statements in a program, execute and check your conclusions.
2. For what values of `a,b,c` will the expressions `a+(b+c)` and `(a+b)+c` evaluate to different values?
3. I want to compute the value of $\binom{100}{6} = \frac{100\times99\times98\times97\times96\times95}{1\times2\times3\times4\times5\times6}$. I have many choices in performing this computation. I can choose the order in which to perform the multiplications and divisions, and I can choose the data type I use for representing the final and intermediate results. Here is a program which does it in several ways. Guess which of these are likely to give the correct answer, nearly the correct answer, or the wrong answer. Then run the program and check which of your guesses are correct.

```
main_program{
  int x = 100 * 99 * 98 * 97 * 96 * 95/ (1 * 2 * 3 * 4 * 5 * 6);
  int y = 100/1 * 99/2 * 98/3 * 97/4 * 96/5 * 95/6;
  int z = 100/6 * 99/5 * 98/4 * 97/3 * 96/2 * 95/1;

  int u = 100.0 * 99 * 98 * 97 * 96 * 95/ (1 * 2 * 3 * 4 * 5 * 6);
  int v = 100.0/1 * 99/2 * 98/3 * 97/4 * 96/5 * 95/6;
  int w = 100.0/6 * 99/5 * 98/4 * 97/3 * 96/2 * 95/1;

  cout << x << " " << y << " " << z << endl;
  cout << u << " " << v << " " << w << endl;
}
```

4. What will be the effect of executing the following code fragment?

```
float f1, f2,centigrade=100;
f1 = centigrade*9/5 + 32;
f2 = 32 + 9/5*centigrade;
cout << f1 << ' ' << f2 << endl;

char x = 'a', y;
y = x + 1;
cout << y << ' ' << x + 1 << endl;
```

5. Write a program that reads in distance d in inches and prints it out as v miles, w furlongs, x yards, y feet, z inches. Remember that a mile equals 8 furlongs, a furlong equals 220 yards, a yard is 3 feet, and a foot is 12 inches. So your answer should satisfy $d = (((8v + w) \cdot 220 + x) \cdot 3 + y) \cdot 12 + z$, and further $w < 8, x < 220, y < 3, z < 12$.

6. What is the state of the computer, i.e. what are the values of the different variables and what is on the screen, after 4 iterations of the loop of the spiral drawing program of Section 3.4? Write down your answer without running the program. Then modify the program so that it prints the values after each iteration and also waits a few seconds so you can see what it has drawn at that point. Run the modified program and check whether what you wrote down earlier is correct.

7. Write a program that prints the arithmetic sequence $a, a + d, a + 2d, \ldots, a + nd$. Take a, d, n as input.

8. Write a program that prints out the geometric sequence $a, ar, ar^2, \ldots, ar^n$, taking a, r, n as input.

9. Write a program which reads in `side`, `nsquares`, `q`. It should draw `nsquares` as many squares, all with the same center. The sidelength should increase by `q` starting at `side`. Repeat with the modification that the sidelength should increase by a factor `q`.

10. Write a program which prints out the squares of numbers from 11 to 99.

11. What does the following program draw?

```
main_program{
  turtlesim();
  int i=0;

  repeat(30){
    left(90);
    forward(200*sine(i*10));
    forward(-200*sine(i*10));
    right(90);
    forward(10);
    i++;
  }
  wait(5);
}
```

12. The ASCII codes for the digits 0 through 9 are 48 through 57. Suppose in response to the third statement below, the user types in two digits. The ASCII codes for the digits will then be placed in `p,q`. You are to fill in the blanks in the code such that `dig1` gets the value of the digit in `p` (not the value of its ASCII code), and similarly `dig2` should get the value of the digit in `q`. Finally, the integer `n` should contain the value of the number in which `p` is in the tens place and `q` in the units place.

```
char p,q;
int dig1,dig2,n;
cin >> p >> q;   // equivalent to cin >> p; cin >> q;
dig1 = ...
dig2 = ...
n = ...
```

For example, if the user typed '1', '2', then `p,q` will contain the values 49, 50. At the end we would like `dig1`, `dig2`, `n` to be respectively 1, 2, 12.

13. Write a program that takes as input the coordinates of two points in the plane and prints out the distance between them.

14. Write the program for computing the maximum of numbers as suggested initially in Section 3.4.1, i.e. the one in which `maximum` was to be initialized to the value `(- numeric_limits<float>::max())`. Does this program behave identically (i.e. give the same result for the same inputs) to the program given at the end of the Section 3.4.1? If you think the programs behave differently, state the inputs for which the programs will behave differently.

15. What does the following program compute?

```
double x;
int n;
cin >> x >> n;
repeat(n){
  x = x * x;
}
```

16. Draw a smooth spiral. The spiral should wind around itself in a parallel manner, i.e. there should be a certain point called "center" such that if you draw a line going out from it, the spiral should intersect it at equal distances as it winds around.

CHAPTER 4

A Program-design Example

In this chapter, we will use what we have learned so far to write a slightly complex program. The process of writing it will illustrate some important ideas in designing programs.

The problem we consider is of finding the value of e, the base of the natural logarithm. The number e can be written as the following infinite series.

$$e = \lim_{n\to\infty} \frac{1}{0!} + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \cdots + \frac{1}{n!}$$

It turns out that the terms of the series decrease very fast, so that you get a good approximate value by evaluating the series to a few terms.

The first step in writing a program is to write down a specification, by which we mean a precise description of what is the input to the problem, and are the outputs, and what it means for the output generated by the program to be correct. Next comes the step of designing the program itself. After that, you typically test the program, i.e. compile and run it on some inputs to see if it works correctly. It might so happen, that the program makes a mistake, in which case you need to go back and try to find what went wrong. This step is often called *debugging*, where a *bug* is a common euphemism for a programming error. In addition to testing the program, you may formally *reason* to yourself that your program is correct.

We consider these steps next.

4.1 SPECIFICATION

As mentioned above, the specification for a program states clearly what the input and the output of the program will be. For our program to compute e, what should the input be? A natural possibility is to ask the user to state how much of the series should be summed.

> **Input to e computation program:** Integer n, where $n \geq 0$.
> **Output from e computation program:** $1/0! + 1/1! + \ldots + 1/n!$.

You may have thought that the specification for our program is "obvious". However, note that the input n could have been interpreted as the number of terms to which the series should be summed, in which case the output would have to be $1/0! + 1/1! + \ldots + 1/(n-1)!$. So there appear to be two "obvious" ways of specifying the input. This may often happen. In such cases, it doesn't really matter

what we choose, so long as we clearly state what we have chosen. A second point to be noted is that we have made a remark about the input being required to be non-negative. In professional programs, you are expected to check first whether the valid inputs are specified by the user. In this small example, we will ignore this issue, but it is a point you should note. It is a good idea to tell the user of the program what is a valid input and what isnt.

4.1.1 Examples

"Wait a minute", I would say. "Is there a particular example of this general problem?"
RICHARD FEYNMAN, SURELY YOU'R JOKING MR. FEYNMAN

It is good to write down some examples of the specification, i.e. for some specific input values what the output values ought to be. For our program, you may write: For input 0, the output should be 1. For input 1, the output should be 2, for input 2, the output should be 2.5. Writing down examples forces you to check that you are being alert while writing the specification. Indeed, you may write the specification as above, but in your mind might still be thinking that n denotes the number of terms to be added. When you make up an example, your confusion will vanish.

Also, when your program is written, these examples can be used to test it.

4.2 PROGRAM DESIGN

The first, extremely important, idea in designing programs is to think about how you would solve the problem using a paper and pencil, without computers. Once you are clear in your mind how to solve a problem using paper and pencil, it often suffices to mimic the solution on a computer.

Quite likely, you have already tried to solve the problem using paper and pencil, if you tried to construct examples as suggested in Section 4.1.1. You probably computed the terms of the series, and added them together as you went along. It is probably a good idea to imagine yourself doing the calculation for a large value of n, say $n = 10$. In this process, you will perhaps see that there is a general pattern, and you might also see how to do the calculation efficiently. In particular, suppose you have just calculated the value of the term $1/3!$, and then you go to the next term, $1/4!$. Calculating $1/4!$ involves dividing 1 by the numbers from 1 to 4, but of these divisions, you just did the divisions from 1 to 3 when you calculated $1/3!$. So you can get the value of the term $1/4!$ simply by dividing $1/3!$ by 4. So to calculate any term $1/t!$, you do just one additional division: you take the term $1/(t-1)!$ that you previously computed, and divide that further by t.

Next, you need to figure out if there a repetitive pattern in your calculations. If you find that you are performing similar steps repeatedly, you could perhaps put those steps in a `repeat` statement. Indeed, there is a pattern. The process of calculating the term $1/t!$ is very similar to the process of computing $1/(t-1)!$. So it would seem that you should indeed have a `repeat` loop. We want to calculate the sum $1/0! + 1/1! + \ldots + 1/n!$, which has $n+1$ terms, so it needs n additions. So presumably, we will use n iterations of a `repeat` loop. And our goal will be that we should have $1/0! + 1/1! + \ldots + 1/t!$ calculated after t iterations of the loop. Thus, in the tth iteration we will calculate $1/t!$, which we will then add to the sum.

The next step is to decide what variables we need in the program. This step is a bit tricky. When you imagine yourself solving a problem using a paper and pencil, you just keep on doing the additions or multiplications (or divisions in this case) using more paper as necessary. You may have written a lot of numbers on the paper as you worked, but that doesnt mean you need a separate variable for holding each number that you might have written. The key question to ask is: what data do we need at the

beginning of the tth iteration in order to perform the work that we planned for the iteration? We need a variable to hold each such piece of data.

Clearly, we need to remember the sum of the series calculated so far. Thus, we should have a variable `result` in which the sum computed so far will be held. This variable should be of a floating type. It is customary to use high precision, and so we will use the type `double`. Further, we said that to calculate $1/t!$ we need the value $1/(t-1)!$ which we calculated in the previous iteration. So we will have a variable `term` in which we will expect to hold the value $1/(t-1)!$ at the beginning of the tth iteration. Finally in the tth iteration we need to divide by t to get the new term value that needs to be added to `result`. In other words, we need to know which iteration just finished. So we will use a variable `i` which will hold the value t during the tth iteration. What we have decided about the program can be summarized as the following sketch.

```
main_program{
  int n; cin >> n;
  int i = ...;                        // we fill in the blanks later.
  double result = ..., term = ...;

  repeat(n){
    // Need code to calculate 1/0!+1/1!+...+1/t! in the tth iteration.
    // At the beginning of the tth iteration
    // i = t, term = 1/(t-1)!, result = 1/0!+1/1!+...+1/(t-1)!
  }
  cout << result << endl;
}
```

Our plan, as stated in the comment, is actually enough for us to complete the program. Imagine executing the program and entering the loop for the first time. In our plan, this corresponds to $t = 1$. So we want the variable `i` to be 1. Thus, we must initialize `i` to 1 when we create it. Second, we want `term` to have the value $1/(t-1)! = 1/0!$, since $t - 1 = 0$. Thus, we need to initialize `term` also to $1 = 1/0!$. The variable `result` must have the value equal to the everything in the sequence sum until the term $1/1!$. Thus we should initialize `result` also to $1/0! = 1$.

Next, we decide precisely what we need to do inside the loop. Imagine we are executing the tth iteration of the loop. Our idea was to have `result` get the value $1/0! + \ldots + 1/t!$ in the tth iteration. Assuming everything has gone according to our plan so far, we will have the values $1/0! + \ldots + 1/(t-1)!$ in `result` and $1/(t-1)!$ in `term`. So we need to add $1/t!$ to `result`. But $1/t! = (1/(t-1)!)/t$. Thus `term/i` will have the value $1/t!$. Thus we should have the following statement inside the loop:

```
result = result + term/i;
```

This will leave in `result` the value that we planned. Is this enough? No, in order to stick to our plan, in the $t+1$th iteration, we will need to have the value $t+1$ in the variable `i`, and the value $1/t!$ in the variable `term`. This can be acheived by writing inside the loop:

```
term = term/i;
i = i + 1;
```

```
/************************************************
   Program to calculate e.
   Calculates 1/0! + 1/1! + ... + 1/n!, for input n.  n >= 0.

   Abhiram Ranade, 24/2/13
************************************************/
main_program{
  int n; cin >> n;      // the last term to be added is 1/n!

  int i=1;              // counts iterations of the loop
  double term = 1.0;    // for holding terms of the series
  double result = 1.0; // Will contain the final answer

  repeat(n){ // Plan: When entering for the tth time, t = 1,2,..,n
             // i = t, term = 1/(t-1)!, result = 1/0!+...+1/(t-1)!
    result = result + term/i;
    term = term/i;
    i = i + 1;
  }
  cout << result << endl;
}
```

Fig. 4.1 *A program to compute* e

The complete program is given in Figure 4.1. Note that we could have written the three statements inside the loop as `result += term/i; term /= i; i++;`. Indeed this is oftern preferred. Note that the loop body could also have been:

```
term = term/i;
result = result + term;
i = i+1;
```

In this the new value of `result` is calculated using the old value of `result` and the new value of `term`. In the version in Figure 4.1, the new values of all variables are calculated from the old values of all variables. Some may therefore find the code in Figure 4.1 to be simpler.

4.2.1 Testing

The next step is to run the program and test if it really works. I compiled and ran this program supplying the values 1,2,3,4 for `n`, and it did print out the answers 2, 2.5, 2.667, 2.70833. If you did the calculation by hand as suggested earlier, you will realize that these values are indeed correct. You could also try some large value for `n`, say 10. When I did this, I got the result 2.71828. Since e is a famous number, you should be able to get its value from textbooks, and you will see that 2.71828 is the often quoted value.

4.2.2 Correctness Proof

Testing is one way to check if your program is correct. However, testing does not really give you a complete guarantee of correctness. You know what the program does for the input values that you checked; but how can you be sure that the program will not give a wrong answer on other values?

One way to be sure is to *prove* that the program is correct. This is often not practical for large programs. However, a proof can be written for our small program for computing e. We do this next. It will have some important lessons in general too.

You may be saying at this point, "But our program is *obviously* correct, after all didn't we *design* it so that the variable `result` has the desired value?" Unfortunately, it isnt so simple: mistakes can creep in at any stage. Let me confess that when I first wrote the program of Figure 4.1, I forgot to initialize the variable `i`. As a result, I was getting very strange answers. Forgetting to initialize a variable is a "silly" mistake, but it is very easy to make silly mistakes! This is an important humbling lesson that programming teaches you. Note that if our program is doing something serious, say controlling an aircraft in flight, a mistake of any kind can cause a crash. So we must learn to avoid even silly mistakes.

So as a cross check, we will try to anyway prove the correctness of the program after we have finished writing it. In this proof, we will basically check whether our program is adhering to our plan, i.e. we confirm whether the variables indeed take the values we expect them to take. The proof is based on mathematical induction. The induction hypothesis is what we stated as our plan.

❑ *Induction Hypothesis*

The values of `i, term, result` on the tth entry to the loop are respectively

$$t, \quad 1/(t-1)!, \quad \frac{1}{0!} + \ldots + \frac{1}{(t-1)!}$$

For the base case, we consider $t = 1$, i.e. the values on the first entry. Substituting $t = 1$ in the values in the Induction Hypothesis, we see that we want `i, term, result` to be all 1. But the code before the loop indeed initializes all these variables to 1. Thus, we have established the base case. Note that when you do this part of the proof, you will discover if you indeed forgot to initialize any variable.

Next we will assume that the Induction Hypothesis is true on the tth entry, and show that it must also hold on the $t + 1$th entry. Thus, we need to prove:

❑ *Induction Step*

The values of `i, term, result` on the $t + 1$th entry to the loop are respectively

$$t+1, \quad 1/t!, \quad \frac{1}{0!} + \ldots + \frac{1}{t!}$$

To prove this, let us examine what happens during the tth iteration. First, we execute `result = result + term/i;`. At the beginning of the tth iteration, we know by assumption that `result` has the value $1/0! + \ldots + 1/(t-1)!$, and term the value $1/(t-1)!$, and i the value t. Thus, the statement will cause $1/(t-1)!$ to be first divided by t, and then added. Thus, `result` will get the value $1/0! + \ldots + 1/(t-1)! + 1/t!$. This value will not change during the rest of the iteration, and hence it will stay at the time of entering the loop for iteration $t + 1$. Thus, we have proved the last part of the induction step. In the tth iteration we next execute `term = term/i;`. The value of `term` and `i` on the tth entry are $1/(t-1)!$ and t respectively. Thus, this statement would cause `term` to

become $1/t!$. This value will not change during the rest of the iteration, and hence we have proved the second part of the induction step. The last statement executed in the loop is `i=i+1;`. This will cause `i` to increase to $t+1$. This is also the value we needed for the $t+1$th iteration. Thus, we have proved the induction step. The induction is complete.

Once we have proved the induction hypothesis, we know what the program will print. The program will execute n iterations, where n was the value we typed in response to the statement `cin » n;`. Thus, the program will print the value the variable `result` at the end of n iterations. We have argued above that this value will be $1/0! + \ldots + 1/n!$. Thus, we have in fact proved that the program will work correctly for all n.

4.2.3 Invariants

We could have characterized the values taken by the variables `i, term, result` in the following manner:

> At the beginning or at the end of any iteration of the `repeat` loop, let $i, term, result$ be the values of the variables `i, term, result`. Then these values satisfy the following relationships.
>
> $$term = 1/(i-1)!, \qquad result = 1/0 + 1/1! + \ldots + 1/(i-1)!$$

Notice that this statement is independent of which iteration is being considered. Such statements are called loop invariants, and these are more natural in other contexts (Section 7.8.1).

4.3 DEBUGGING

Unless you are one of the lucky/clever few, it is inevitable that the programs you write will not work on the first try. You will quite likely forget a semicolon or make some other mistake because of which the compiler will complain. The compiler will usually state the line number in which the error is present, so generally it will be easy to correct your mistake. But even after your program compiles correctly, it is possible that it will produce the wrong answers. What do you do in that case?

Clearly, you must go over your entire process of design. Did you get the specifications right? Have you forgotten to initialize a variable? After these basic checks, you should turn to the plan you wrote. In the plan, you have essentially written down how you expect the variable values to change in the different iterations. So consider putting down print statements which print out the values of the variables in each iteration. Then you will have to calculate by hand if they are as you expect them to be. We will see some shortcuts for this later, but basically this is what you need to do.

You may be tempted to say that your program is correct but the computer is making a mistake—but computers make mistakes so rarely that this possibility can be safely ignored.

4.4 COMMENTS IN THE CODE

We have remarked earlier that programs should be written not only so that they can be compiled and executed to solve problems, but also so that they can be easily understood by other programmers.

There are several ways to make a program easier to understand. Most of these ways involve putting in appropriate comments in code. For example, the specifications should be written down in the

comments. Another way is to choose good names for the variables so that the names convey the purpose. In addition, you could write a comment along with the definition of the variable.

A very important aid to understandability is explaining the plan for a loop. The plan should be described in enough detail, so that it should be possible to understand the progress made in each iteration towards the final goal. Later on we will suggest other (related) ways of explaining loops, e.g. invariants and potential (Section 7.8).

4.5 CONCLUDING REMARKS

It is useful to summarize the main steps in designing a program.

Typically we start with an English language, semi-precise statement of the problem. From this we first generate a precise specification, with clear characterization of the input and output. The general relationship between the input and the output must be stated, and also some examples must be given.

As to designing the program, one strategy is to first try to solve the problem using a paper and pencil, without a computer. Then we can mimic the paper-pencil solution on a computer. Note that there is a difference between being able to solve a problem and consciously knowing how you solve it. By "consciously knowing" we mean things such as being able to break up the solution into a sequence of actions, and also identifying patterns in the sequence. This is not difficult, but requires some practice and introspection. A related issue is to be able to see "what do you need to do in general". In the computation of e, we needed to say that "in general, for any n, we need to have n iterations of the loop". This is pretty much what you do in high school algebra when you say things such as "if a pen costs Rs. 5, then n pens cost Rs $5n$". The ability to state things in general is crucial to writing programs!

Next you need to identify repetitive patterns in the computation, decide what variables to use, write down an overall plan and then write the actual code.

Testing your program is extremely important. We will say more on the subject later. However, for now, try testing on many values. As you can see, it is useful to work out what results you expect using pencil and paper, at least for a few cases.

We also gave an introduction to the process of proving the correctness of programs. Proving programs to be correct turns out to be too tedious for large programs. However, for small programs, proving correctness is very useful, and you will see several examples of it in the book. When you prove a program, you are basically reasoning about how values are assigned to the variables in the program so that the program slowly but steadily makes progress towards computing what it needs to. This progress is made precise in the plan (or *invariant* as we will discuss later on) that we wrote down. Even if you don't bother to prove your programs correct, we strongly recommend that you write the plan for each non-trivial loop in your program. Just the act of writing the plan in detail will help you to get a correct program. The plan must be placed in the program as comments. This will also make your program more understandable to others who might read it. Often, you can first write the plan and then the code, as we just did.

Do everything you can that will increases your confidence that your program is correct. Remember, a wrong program is not just useless, it is potentially *dangerous*.

A Note on Programming Exercises

Programming exercises form a big part of learning to program. Programming cannot be learnt just by reading: practice is extremely important. So please write as many programs as possible. Follow the guidelines suggested in this chapter while writing programs.

EXERCISES

1. Write a program to compute the value of

$$D(r) = \sum_{k=0}^{r} (-1)^k \frac{r!}{k!}$$

Incidentally, $D(r)$ is the number of ways in which the numbers 1 through r can be arranged in a sequence such that i is never in the ith position, for all i.

2. Here is an infinite product which can be shown to approach $2/\pi$ as the number of terms increases.

$$\frac{2}{\pi} = \frac{\sqrt{2}}{2} \cdot \frac{\sqrt{2+\sqrt{2}}}{2} \cdot \frac{\sqrt{2+\sqrt{2+\sqrt{2}}}}{2} \cdots$$

Write a program that computes the product of the first n terms, where n is specified as input. You will need to specify what values your variables take after some t iterations. For this feel free to write something like "`numerator` has value $\sqrt{2+\sqrt{\ldots+\sqrt{2}}}$ with $\sqrt{}$ appearing t times". Write a proof of correctness.

3. Write a program to approximately compute e^x by adding first 15 terms of the series

$$e^x = \frac{x^0}{0!} + \frac{x^1}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots$$

4. Write a program that computes the value of an nth degree polynomial $A(x) = a_0 + a_1 x + a_2 x^2 + \ldots + a_n x^n$. Assume that you are given n then the value x, and then the coefficients $a_0, a_1, \ldots, a_n$.

5. Evaluate the polynomial, but this time assume that you are given the coefficients in the order $a_n, a_{n-1}, \ldots, a_0$.

6. Figure 4.2 gives two programs to compute e. One of them is incorrect. Find which one. For the correct program, give appropriate invariants and prove its correctness.

```
main_program{
  int n, fac=1, i=2;
  double e=1.0;
  cin >> n;

  repeat(n){
    e = e + 1.0/fac;
    fac = fac * i;
    i = i + 1;
  }
  cout << e << endl;
}
```

(a)

```
main_program{
  int n, fac=1, i=1;
  double e=1.0;
  cin >> n;

  repeat(n){
    e = e + 1.0/fac;
    fac = fac * i;
    i = i + 1;
  }
  cout << e << endl;
}
```

(b)

Fig. 4.2 *One of these programs is incorrect.*

7. Write a program which multiplies an n digit number M by a 1-digit number d, where n could be large, e.g. 1000. The input will be given as follows. First the user gives d, then n and then the digits of M, starting from the least significant to the most significant. The program must print out the digits of the product one at a time, from the least significant to the most significant.

The program you write will likely perform about n multiplication operations and a similar number of other operations. There is a more efficient way of writing this program, i.e. using fewer operations for multiplying the same numbers M, d. Hint: Ask the user to give several digits of M at a time.

CHAPTER 5

Simplecpp Graphics

The graphics commands we introduced in Chapter 1 are fun, but quite limited. The more general graphics system that we discuss in this chapter has many other features:

- Ability to have several turtles on screen simultaneously, moving and drawing as desired.
- Ability to create other shapes, e.g. lines, rectangles, circles, polygons and text on the screen and move these shapes as desired. The shapes also have pens, so they can also draw on the screen if needed.
- Ability to change attributes such as colour, size of the various shapes.
- Ability to draw lines on the screen by specifying coordinates rather than have a turtle trace lines while moving. Likewise shapes can be made to move by giving the coordinates on the screen, rather than always having to be specified relative to the position and the orientation of the object in question.
- Elementary graphical input. The user can click on the graphics window and the program can wait for such clicks and get the click coordinates.

We will discuss these features in this chapter. Following that, we will build two example programs. The first is somewhat simple: plotting the trajectory of a projectile as it moves under the influence of gravity. The second is more involved. In this the user can click a set of points on the screen, and the program will draw the best fit straight line through the points, under a certain measure of goodness.

In a later chapter, we will present some additional graphics related features.

5.1 OVERVIEW

To access the more general graphics facilities, it is more convenient to use the command:

```
initCanvas();
```

rather than `turtleSim()`. This opens a window, but does not create a turtle at its center. Commands `canvas_width(), canvas_height()` return the width and height of the canvas in pixels. You may also invoke the command as

```
initCanvas(name,w,h)
```

where `name` is a quoted string meant to be the name given for the canvas, and `w,h` should indicate the width and height you desire for the canvas window. This form is also available for `turtleSim`.

5.1.1 y-axis Goes Downward!

A coordinate system is associated with the canvas window. You may find it slightly unusual: the origin is at the top left corner, and x-coordinates increase rightward, and y coordinates downward. The coordinates are measured in pixels. Note however that internally, Simplecpp considers the coordinates of objects to be real numbers of type `double`. These real coordinates are converted to integers only when needed for the purpose of displaying the objects.

5.2 MULTIPLE TURTLES

We can create multiple turtles very easily, by writing:

```
Turtle t1,t2,t3;
```

This will create three turtles, respectively named `t1, t2, t3` at the center of the window created using `initCanvas()`. Yes, the turtles will all be at the center, stacked one on top of the other. We next see how we get them untangled.

The basic idea is: any command you used in Chapter 1 to affect the turtle will also work with these turtles, but you must say which turtle you are affecting. For this, you must write the command following the name of the turtle, the two joined together by a dot: ".". Thus, to move turtle `t1` forward by 100 steps, we merely write

```
t1.forward(100);
```

Likewise, to turn `t2`, we would write

```
t2.left(90);
```

The same thing applies to other commands such as `right, penUp, penDown`.

Here is a program which will use 3 turtles to draw 3 octagons, aligned at 120 degrees to each other.

```
main_program{
  initCanvas();
  Turtle t1, t2, t3;

  t2.left(120);
  t3.left(240);

  repeat(8){
    t1.forward(100);
    t2.forward(100);
    t3.forward(100);

    t1.left(360.0/8);
    t2.left(360.0/8);
    t3.left(360.0/8);
  }
  wait(5);
}
```

5.3 OTHER SHAPES BESIDES TURTLES

Three other shapes are allowed besides turtles: circles, axis-parallel rectangles, and straight-line segments. Text is also considered to be a kind of shape. Later in Section 15.2.3 we will define a polygon shape.

5.3.1 Circles

Circles can be created by writing

```
Circle c1(cx,cy,r);
```

Here, `cx,cy,r` must be numerical expressions which indicate the radius of the circle, and the x and y coordinates of its center. The created circle is named `c1`.

5.3.2 Rectangles

An axis-parallel rectangle is defined as follows

```
Rectangle r1(cx,cy,Lx,Ly);
```

where `cx,cy` should give the coordinates of the center, and `Lx,Ly` the width and height respectively. The created rectangle has the name `r1`.

5.3.3 Lines

A line segment can be defined as

```
Line line1(x1,y1,x2,y2);
```

This creates a line named `line1` where `x1,y1` are the coordinates of one endpoint, and `x2,y2` the coordinates of the other.

5.3.4 Text

If we want to write text on the screen, it is also considered a kind of shape. The command

```
Text t1(x,y,message);
```

in which `x,y` are numbers and `message` is a text string can be used to write the message on the screen. So you might use the command `Text txt(100,200,"C++");` to write the text `C++` on the screen centered at the position (100,200). Another form is

```
Text t2(x,y,number);
```

Here, `number` can be a numerical expression. The value of the expression at the time of execution of this statement will comprise the text. It will be centered at the coordinates `(x,y)`.

The command `textWidth` can be used to find the width of the given text in pixels. For example `textWidth("C++")` returns the width of the text "C++" when drawn on the canvas. The command `textHeight()` returns the height in pixels. Thus, the following piece of code can be used to write text and put a snugly fitting box around it.

```
Text t(100,100,"C++ g++");
Rectangle R(100,100,textWidth("C++ g++"),textHeight());
```

If for some reason you wanted to know by how much the lower part of "g" descends below the line on which the text gets written, you can know this using the command `textDescent()`.

5.4 COMMANDS ALLOWED ON SHAPES

Each shape mentioned above can be made to move forward and rotate, except for `Text` shapes, which cannot be rotated. Each shape also has a pen at its center which can be either up or down.

In addition, for any shape `s`, we have the commands

```
s.moveTo(x,y);
s.move(dx,dy);
```

where the former moves the shape to coordinates `(x,y)` on the screen, and the latter displaces the shapes by `(dx,dy)` from its current position.

You can change the size of a shape (except for text) also. Every object maintains a scale factor, which is initially set to 1, based on which its size is displayed.

```
s.scale(relfactor);
s.setScale(factor);
```

Here, `relfactor, factor` are expected to be expressions of type `double`. The first version multiplies the current scale factor by the specified `relfactor`, the second version sets the scale factor to `factor`.

Scaling is not supported on `Text`.

You can also decide whether a shape `s` is to appear in outline, or it is to be filled with some color. For this, the following command can be used.

```
s.setFill(v);
```

Here, `v` must be specified as `true` or `false`. If `v` is specified as `false`, the shape will be drawn in outline, other it will be *filled* with its current colour.

The colour of a shape can be changed by writing:

```
s.setColor(color);
```

where `color` is specified for example, as `COLOR("red")`. Note that merely specifying `"red"` will not work. Instead of red, other standard color names, e.g. blue, green, yellow, white, black can be used. Use all lowercase letters. Alternatively, you may specify the color by giving intensities of three primary colors, red, green, blue respectively, by writing `COLOR(redVal, greenVal, blueVal)`. The three values should be numbers between 0 and 255. As you may guess, red and blue together give purple, while red and green give yellow.

You may hide or unhide a shape `s` using the commands

```
s.hide();
s.show();
```

respectively.

5.4.1 Rotation in Radians

The `left, right` commands of Chapter 1 required angles to be supplied in degrees. However, most programming languages including C++ prefer angles to be represented in radians. For this a `rotate` command is provided. Thus if `s` is a shape you may write `s.rotate(angle)` where `angle` must be the angle in radians, measured clockwise.

Rotation is not supported on `Text`.

5.4.2 Tracking a Shape

As the program executes, you may move shapes or rotate them or scale them. You can of course keep track of the position, orientation, scale factor yourself, but you do not need to; `simplecpp` does it for you. The following commands will return the x coordinate, the y coordinate, the orientation, and the scale factor respectively.

```
s.getX()
s.getY()
s.getOrientation()
s.getScale()
```

You may print the values by writing `cout << s.getOrientation();` and so on, or you may use them in computation. The `getOrientation` command will return the angle made by the shape with the positive x-axis, measured clockwise.

5.4.3 Imprinting on the Canvas

Suppose `s` is a shape. Then the following command causes an image of the shape to be printed on the canvas, at the current position of `s`.

```
s.imprint();
```

After this, the shape might move away, but the image stays permanently. You can print as many images of a single shape as you desire. The new image overwrites older images, if any. The command works with all shapes `s`.

If you merely want to draw lines on the screen for some reason (e.g. Section 20.3) an additional command is also provided.

```
imprintLine(x1,y1,x2,y2,color)
```

or

```
imprintLine(x1,y1,x2,y2)
```

This will draw a line between the points `(x1,y1)` and `(x2,y2)`, of colour `color`. If `color` is not given, then the line will be black. You could have got the same effect by creating a line and then calling `imprint` on it; however, the command `imprintLine` is much faster. The speed is sometimes important, as in the application of Section 20.3.

5.4.4 Resetting a Shape

For each shape, except `Turtle`, a `reset` command is provided. This command takes the same arguments as required for creation, and recreates the shape using the new values. For example, you could have

```
Circle c(100,100,15);
wait(1);
c.reset(100,100,20);
```

This would have the effect of expanding the circle.

5.5 CLICKING ON THE CANVAS

The command `getClick()` can be used to wait for the user to click on the canvas. It causes the program to wait until the user clicks. Suppose the user clicks at a point (x, y) on the screen. Then the value $v = 65536 \times x + y$ is returned by the command. Note that the click is considered to be happening on some pixel, i.e. the coordinates x, y of the click position are integers. The value returned by `getClick()` is also of type `int`.

Note that standard computer screens will have at most a few thousand pixels along the height and along the width. Thus, the click coordinates x, y will at most be a few thousand. Thus $x, y < 65536$. So if you are given $v = 65536 \times x + y$, then you can recover x, y by noting that

$$x = \lfloor v/65536 \rfloor, \qquad y = v \bmod 65536$$

As an example, the following program waits for the user to click, and then prints out the coordinates of the point at which the user clicked.

```
main_program{
  int clickPos;

  initCanvas();

  clickPos = getClick();

  cout << "Click position: ("
       << clickPos/65536 <<", "      // integer division: truncates.
       << clickPos % 65536 <<")\n";
}
```

By the way, $65536 = 2^{16}$; so the x coordinate comprises of the more significant 16 bits of the value returned by `getClick`, and the y coordinate the least significant 16.

5.6 PROJECTILE MOTION

We will now write a program that simulates the motion of a projectile. Suppose that the projectile has initial velocity 1 pixel per step in the x-direction, and -5 pixels per step in the y-direction (note that the y coordinate grows downward, so this is upward velocity). Let us arbitrarily fix the gravitational acceleration to be 0.1 pixels/step^2. For simplicity, assume that the velocity only changes at the end of each step: at the end of each step 0.1 gets added to the y-component velocity.

```
#include <simplecpp>

main_program{
  initCanvas("Projectile motion", 500,500);

  int start = getClick();

  Circle projectile(start /65536, start % 65536, 5);
  projectile.penDown();

  double vx=1,vy=-5, gravity=0.1;

  repeat(100){
    projectile.move(vx,vy);
    vy += gravity;
    wait(0.1);
  }
  wait(10);
}
```

The program waits for the user to click. It then places a projectile, a `Circle` at the click position. Then it moves the projectile as per its velocity. The pen of the projectile is put down so that the path traced by it is also seen. The projectile is moved for 100 steps.

5.7 BEST-FIT STRAIGHT LINE

Suppose you are given a set of points $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$. Your goal is to find a line $y = mx + c$ which is the closest to these points. We will see a way to do this assuming a specific definition of "closest".

A natural definition of the distance from a point to a line is the perpendicular distance. Instead, we will consider the "vertical" distance $y_i - mx_i - c$. We will try to minimize the total distance of all points from our line; actually, since the quantity $y_i - mx_i - c$ can be positive or negative, we will instead minimize the sum E of the squares of these quantities, i.e.

$$\min E = \sum_{i=1}^{n} (y_i - mx_i - c)^2$$

Consider E to be a function of m. At the chosen value of m, E must be smallest, i.e. the derivative of E with respect to m must be 0.

$$0 = \frac{d}{dm} \sum_{i=1}^{n} (y_i - mx_i - c)^2 = -2 \sum_{i=1}^{n} x_i (y_i - mx_i - c)$$

The terms of this can be rearranged as an equation in m and c:

$$m \sum_i x_i^2 + c \sum_i x_i = \sum_i x_i y_i \tag{5.1}$$

Likewise, considering E to be a function of c, we must have the derivative of E with respect to be c to be 0.

$$0 = \frac{d}{dc}\sum_{i=1}^{n}(y_i - mx_i - c)^2 = -2\sum_{i=1}^{n}(y_i - mx_i - c)$$

This can also be rewritten as an equation.

$$m\sum_i x_i + nc = \sum_i y_i \tag{5.2}$$

Define $p = \sum_i x_i^2$, $q = \sum_i x_i$, $r = \sum_i x_i y_i$, and $s = \sum_i y_i$. Then Equation (5.1) becomes $pm + qc = r$ and Equation (5.2) becomes $qm + nc = s$. These equations are easily solved symbolically, giving

$$m = \frac{nr - qs}{np - q^2} \qquad c = \frac{ps - qr}{np - q^2}$$

The program is given below. The variable names in it are as per the discussion above.

```
main_program{  // Fit line to set of points clicked by the user.
  cout << "Number of points: ";
  int n; cin >> n;   // number of points to which the
                     // line is to be fit.

  initCanvas("Fitting a line to data",500,500);

  double p=0, q=0, r=0, s=0;
  Circle pt(0,0,0);  // Will be used to show point clicked by user
  repeat(n){
    int cPos = getClick();
    double x = cPos/65536;
    double y = cPos % 65536;
    pt.reset(x,y,5); // Centered at the click position
    pt.imprint();    // Because we will move pt for
                     // subsequent points.

    p += x*x;
    q += x;
    r += x*y;
    s += y;
  }
  double m = (n*r - q*s)/(n*p - q*q);
  double c = (p*s - q*r)/(n*p - q*q);
  Line l(0,c, 500, 500*m+c);
  wait(10);
}
```

5.8 CONCLUDING REMARKS

In this chapter, we studied the basics of Simplecpp graphics. As you saw, it is possible to define shapes such as rectangles, lines and circles and move, scale, and rotate them. You can also write text on the screen, and that too can be moved around but not scaled or rotated. It is also possible to change the colours of all such objects. Finally, we noted that a coordinate system is associated with the graphics window, with the x-axis going to the right, and the y axis going downward.

Using the graphics commands, you should be able to create some interesting animations, like the projectile motion example we discussed.

An important command to be noted is `imprint`. This command is very useful when you want several similar shapes to be drawn on the screen permanently. For this, you can simply create one shape, then move it to different places and imprint it.

If it appears to you that defining shapes is like defining variables, you would be right! Indeed, statement such as

```
Circle c1(100,100,10), c2(300,200,15);
```

not only create circles, but also indeed define two variables, `c1` and `c2`. The commands discussed above are invoked on these variables, and as a result they cause the images on the screen to be changed.

Just as ordinary variables can be defined inside repeat loops, so can these shapes. But just as ordinary variables will get destroyed once we get to the end of the parent block, so will these shapes.

Further, the names of the shapes, `Circle, Rectangle, Line, Turtle` in fact are the data types of the corresponding variables. These are special data types created for Simplecpp. C++ allows creation of data types such as these. We will study this in Chapter 17. For now, you can just use them.

EXERCISES

1. Draw an 8×8 chessboard having red and blue squares. *Hint*: Use the `imprint` command. Use the repeat statement properly so that your program is compact.
2. Plot the graph of $y = \sin(x)$ for x ranging in the interval 0 to 4π. Draw the axes and mark the axes at appropriate points, e.g. multiples of $\pi/2$ for the x axis, and multiples of 0.25 for the y axis.
3. Modify the projectile motion program so that the velocity is given by a second click. The projectile should start from the first click, and its initial velocity should be in the direction of the second click (relative to the first). Also the velocity should be taken to be proportional to the distance between the two clicks.
4. Another idea is to treat the second click to be the highest point reached by the projectile as it moves. For this, you may note that if u_x, u_y are the initial velocities of the projectile in the x, y directions, and g the gravitational acceleration, then maximum height reached is $\frac{u_y^2}{2g}$. The horizontal distance covered by the time the maximum height is reached is $\frac{u_x u_y}{g}$.
5. Modify the projectile motion program to trace the trajectories of the projectile for the same initial velocity and different angles. As you may know, for a fixed velocity, the projectile goes farthest if it is launched at 45 degrees to the horizontal. You should be able to verify this statement using your program.

6. Write a program to produce the following effect. First, a square appears on the screen. Then a tiny circle appears at the center. Slowly, the circle grows until it touches the sides of the square. Then both the circle and the square start shrinking until they vanish.

7. Suppose you are given some observed positions of a projectile. Each position is an (x, y) pair. You are further told that the projectile is surely known to pass through the origin (0,0). Derive the best fit trajectory for the given points, such that it passes through (0,0). For this you will have to adapt the process we followed to fit a straight line.

8. Write a program that accepts 3 points on the canvas (given by clicking) and then draws a circle through those 3 points.

9. Write a program that accepts 3 points, say p, q, r. Then the program draws the line joining p, q. Then the line is rotated around the point r, slowly, through one full rotation. The key question here is how to rotate a line through a point which is not its center. This can be done in two ways. You could calculate the next position of the line, and then `reset` the line to that position. Alternatively, you can observe that a rotation about an external point such as r can be expressed as a rotation about the center and a translation, i.e. a `move`. This will require you to calculate the amount of translation.

10. In this problem, you are to determine how light reflects off a perfectly reflecting spherical surface. Suppose the sphere has radius r and is centered at some point (x, y). Suppose there is a light source at a point (x', y). Rays will emerge from the source and bounce off the sphere. As you may know, the reflected ray will make an angle to the radius at the point of contact equal to that made by the incident ray. Write a program which traces many such rays. It should take r, x, y, x' as input. Of course, in the plane the sphere will appear as a circle. Do not use a turtle to trace the lines, but calculate the coordinates and specify them directly.

11. This is an extension to the previous problem. Extend the reflected rays backward till they meet the line joining the circle center and light source. The points where the rays meet this line can be said to be the image of the light source in the mirror; as you will see this will not be a single point, but the image will be diffused. This is the so-called spherical aberration in a circular mirror.

CHAPTER 6

Conditional Execution

Suppose we want to calculate the income tax for an individual. The actual rules of income tax calculation are quite complex. Let us consider very simplified rules as follows:

> There is no tax if your income is at most ₹ 1,80,000. If your income is between ₹ 180,000 and ₹ 500,000 then you pay 10% of the amount by which your income exceeds ₹ 180,000. If your income is between ₹ 500,000 and ₹ 800,000, then you pay ₹ 32,000 plus 20% of the amount by which your income exceeds ₹ 500,000. If your income exceeds ₹ 800,000, then you pay ₹ 92,000 plus 30% of the amount by which your income exceeds ₹ 800,000.

In the programs we have written so far, each statement was executed once, or each statement was executed a certain number of times, as a part of a repeat block. The statements that we have learned do not allow us to express something like "If some condition holds, then execute a certain statement, otherwise execute some other statement.". This *conditional* execution is required for the tax calculation above.

The main statement which expresses conditional execution is the `if` statement. We will also discuss the `switch` statement, which is sometimes more convenient. We also discuss logical data, and how it can be stored in the `bool` type.

6.1 THE `IF` STATEMENT

We first give the program which calculates the tax, and then explain each statement.

```
main_program{
  float income; // in rupees.
  float tax;    // in rupees.

  cout << "What is your income in rupees? ";
  cin >> income;

  if(income <= 180000) tax = 0;              // first if statement
  if((income >  180000)&&(income <= 500000)) // second if statement
    tax = (income - 180000)* 0.1;
```

```
  if((income >  500000)&&(income <= 800000)) // third if statement
    tax = 32000+(income - 500000)* 0.2;
  if(income > 800000)                        // fourth if statement
    tax = 92000+(income - 800000)* 0.3;

  cout << "Tax is: " << tax << endl;
}
```

This program uses the simple form of the `if` statement, which is as follows.

```
if (condition) consequent
```

In this, the `condition` must be an expression which evaluates to `true` or `false`. We will soon describe how such expressions can be written. In any case, the execution of the `if` statement begins with the evaluation of the `condition` expression. If it evaluates to `true`, then the `consequent`, which can be any C++ statement, is executed. If the `condition` evaluates to `false`, then the `consequent` is ignored. At this point, the execution of the `if` statement ends, and control passes to the next statement in the program. Pictorially, this is often shown in the form of a *flowchart*, Figure 6.1. In this figure, boxes are used to hold statements to be executed, or actions to be performed. It is customary to write conditions inside diamonds. Lines join the boxes and diamonds showing how control can flow. As you can see, after evaluating the `condition`, either the true branch is taken, in which case the `consequent` is executed, or the false branch is taken in which case the control directly goes to the next statement.

The simplest form of `condition` is as follows.

```
exp1 relop exp2
```

where `exp1` and `exp2` are numerical expressions, and `relop` is a relational operator, e.g. `<`, `>`, `<=`, `>=`, `==`, `!=` which respectively stand for less than, greater than, less than or equal, greater than or equal, equal, and not equal. Thus in the first `if` statement in the program, `income <= 180000` is a condition. If during execution, the value of the variable `income` is at most 180000, then the condition evaluates to `true`, and the `condition` is said to *succeed*. If so the `consequent` is executed. Thus, `tax` is set to 0. If `income` is greater than 180000, the `condition` evaluates to `false`, and is said to *fail*. In this case the `consequent` is not executed, i.e. `tax` remains unchanged. Similarly, in the last `if` statement, the condition is `income > 800000`. The consequent here,

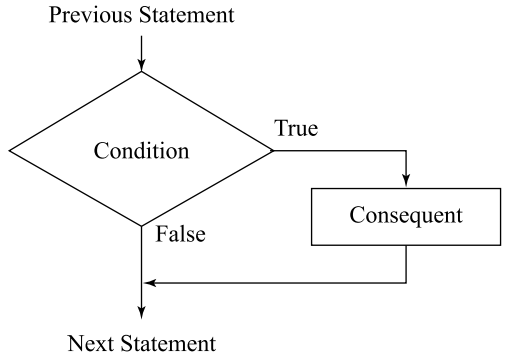


Fig. 6.1 *Flowchart for simple* `if` *statement*

`tax = 92000 + (income - 800000) * 0.3` is executed if and only if the value of `income` is greater than 800000.

It is possible to specify a more complex `condition` in the `if` statement. For example, you may wish to perform a certain operation only if several conditions `condition_1`, `condition_2 ... condition_r` are all true. Thus, our `condition` can be a conjunction (*and*) of all these conditions. This is written as follows.

```
condition_1 && condition_2 && ... && condition_r
```

The characters `&&` should be read as "and".[1] Such a condition is evaluated left to right. If some `condition_i` is found false, then subsequent conditions are not evaluated, and the entire condition is deemed false. The entire condition is deemed true only if all `condition_i` evaluate to true.

In our second `if` statement, we have an example of this. Here, the compound condition will be deemed true only if both the subconditions, `income > 180000` and `income <= 500000` are true. In other words, the compound condition is true only if the income is between 180000 (exclusive) and 500000 (inclusive). Only in this case is the `tax` set to `(income - 180000)* 0.1`, i.e. 10% of the amount by which the income exceeds 180000.

Note that we can have a compound condition which holds if at least one of some set of conditions holds. Such a condition is said to be a disjunction of (sub) conditions and is expressed as:

```
condition_1 || condition_2 || ... || condition_r
```

The characters `||`, constitute the logical *or* operator. To find whether the entire condition is true, we evaluate conditions left to right. If some `condition_i` is found true, then subsequent conditions are not evaluated and the value of the entire condition is deemed true. The value of the entire condition is deemed false if all `condition_i` evaluate false.

Finally, one condition can be the negation of another condition, written as follows:

```
!condition
```

where the condition `!condition` is said to be the negation of `condition`. The condition `!condition` is true if `condition` is itself false, and `!condition` is false if `condition` is true.

We note that conjunctions, disjunctions and negations can be nested inside one another. For example, the second if statement of our program can also be written as

```
if(!((income <=  180000) || (income > 500000)))
  tax = (income - 180000)* 0.1;
```

Notice that `(income <= 180000) || (income > 500000)` is true if income is either less than or equal to 180000 or greater than 500000, i.e. if the income is *not* in the range 180000 (exclusive) and 500000 (inclusive). But the `!` at the beginning negates this condition, so the entire condition is true only if the income is indeed in the range 180000 (inclusive and 500000 (exclusive). But this is the same condition as tested in the second if statement in the program!

It is important to clearly understand how the above program is executed. The execution is as usual, top to bottom. After printing out a message and reading the value of `income`, the program executes the first `if` statement. For this, the condition in it is checked, and then the consequent is executed if

[1] The single character `&` is also an operator, but it means something different, see Appendix C.

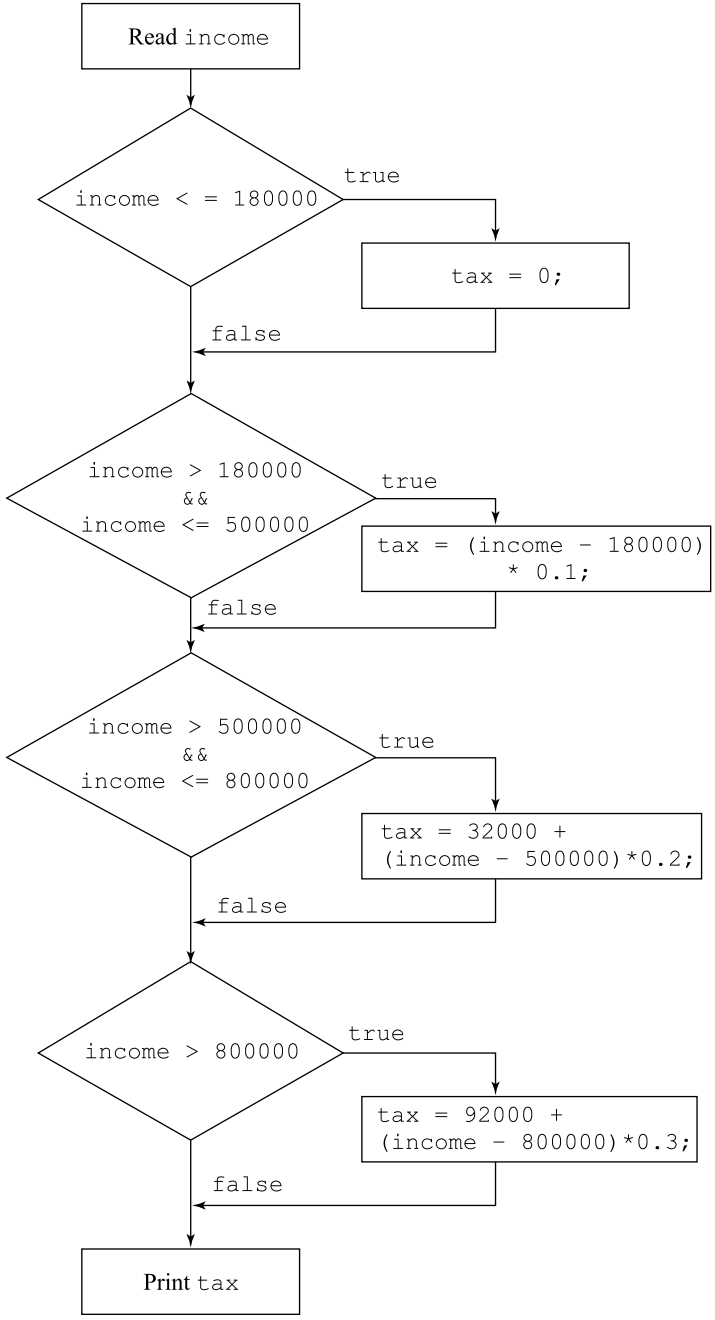


Fig. 6.2 *Flowchart for the first income-tax program*

the condition is true. After this, the second `if` statement is executed. So every `if` statement will be executed; the conditions have been so designed so that the condition in only one `if` statements will evaluate to true, and hence only one consequent statement will be executed. Perhaps the way in which control flows is more obvious in the flowchart of the entire program, shown in Figure 6.2. Note that once we discover a certain condition to be true, e.g. that the income is at most 180000, we know that the other conditions cannot be true. So the natural question arises: why should we even check them?

The more general `if` statement, discussed shortly, allows you to prevent such unnecessary checks. But before discussing that, we discuss the notion of *blocks*.

6.2 BLOCKS

In the `if` statement discussed above, the `consequent` was expected to be a single statement. In general, we might want to execute several statements if a certain condition held, not just one. The block construct helps us in this case.

As discussed earlier, a block is simply a collection of statements that are grouped together in braces, `{` and `}`. By putting statements into a block, we are making a single compound statement out of them. A block can be placed wherever a single C++ statement is required, e.g. as the `consequent` part of the `if` statement. Suppose for example, we want to print a message "This person is in the highest tax bracket." if the income is more than 8 lakhs, as well as calculate the tax, we would replace the fourth `if` statement in the program with the following.

```
if (income > 800000){
    tax = 92000+(income - 800000)* 0.3;
    cout << "This person is in the highest tax bracket." << endl;
}
```

You have already used a block as a part of the `repeat` statement. Let us now note that the general form of the repeat statement is

```
repeat (count) action
```

where `action` is any statement including a block. Thus, we may write

```
repeat (10) cout << "Test." << endl;
```

which will cause the message `"Test."` to be printed 10 times.

6.3 OTHER FORMS OF THE `IF` STATEMENT

The `if-else` statement has the following form:

```
if (condition) consequent
else alternate
```

This statement also begins with the evaluation of `condition`. If it is true, then as before `consequent` is executed. If the `condition` is false, however, then the `alternate` statement is executed. So exactly one out of the statements `consequent` and `alternate` is executed, depending upon whether the `condition` is true or false. This is shown pictorially in the flowchart of Figure 6.3.

The most complex form of the `if` statement is as follows.

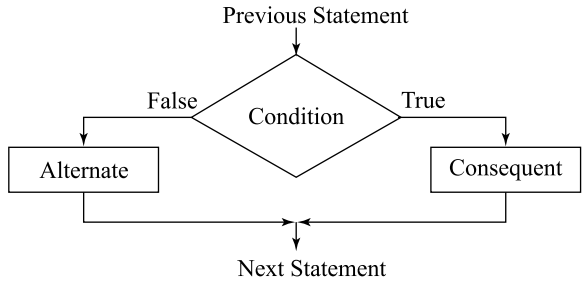


Fig. 6.3 *If statement with else clause*

```
if (condition_1) consequent_1
else if (condition_2) consequent_2
else if (condition_3) consequent_3
...
else if (condition_r) consequent_r
else alternate
```

This statement is executed as follows. First, `condition_1` is checked. If it is true, then `consequent_1` is executed, and that completes the execution of the statement. If `condition_1` is false, then `condition_2` is checked. If it is true, then `consequent_2` is executed, and that completes the execution of the statement. In general, `condition_1, condition_2, ...` are executed in order, until some `condition_i` is found to be true. If so, then `consequent_i` is executed, and the execution of the statement ends. If no condition is found true, then the `alternate` is executed. It is acceptable to omit the last line, i.e. `else alternate`. If the last line is omitted, then nothing is executed if none of the conditions are found true. Figure 6.4 shows a flowchart, for 3 conditions.

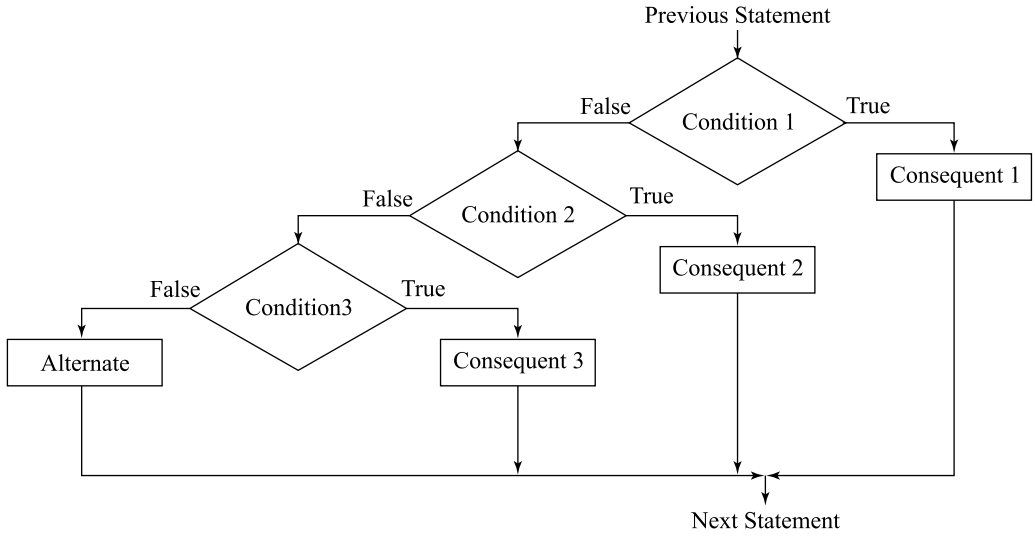


Fig. 6.4 *Most general if, with 3 conditions*

Now we can rewrite our tax calculation program as follows.

```
main_program{
  float income, tax;

  cout << "What is your income? ";
  cin >> income;

  if(income <= 180000) tax = 0;        // new first if
  else if(income <= 500000)            // new second if
    tax = (income - 180000)* 0.1;
  else if(income <= 800000)            // new third if
    tax = 32000+(income - 500000)* 0.2;
  else
    tax = 92000+(income - 800000)* 0.3;

  cout << "Tax is: " << tax << endl;
}
```

Notice that this program contains only three conditions, rather than four as in the previous program. This is because if all the three conditions are false, we know that the income must be bigger than 800000. Thus even without checking this condition we can directly set the tax to `92000+(income - 800000)* 0.3`.

Also note that the second and third conditions are much simpler! In the first program, we checked if the income was larger than 180000 and at most 500000. In the new program, we know that the "new second if" is executed only if the condition of "new first if" failed, i.e. if the income was greater than

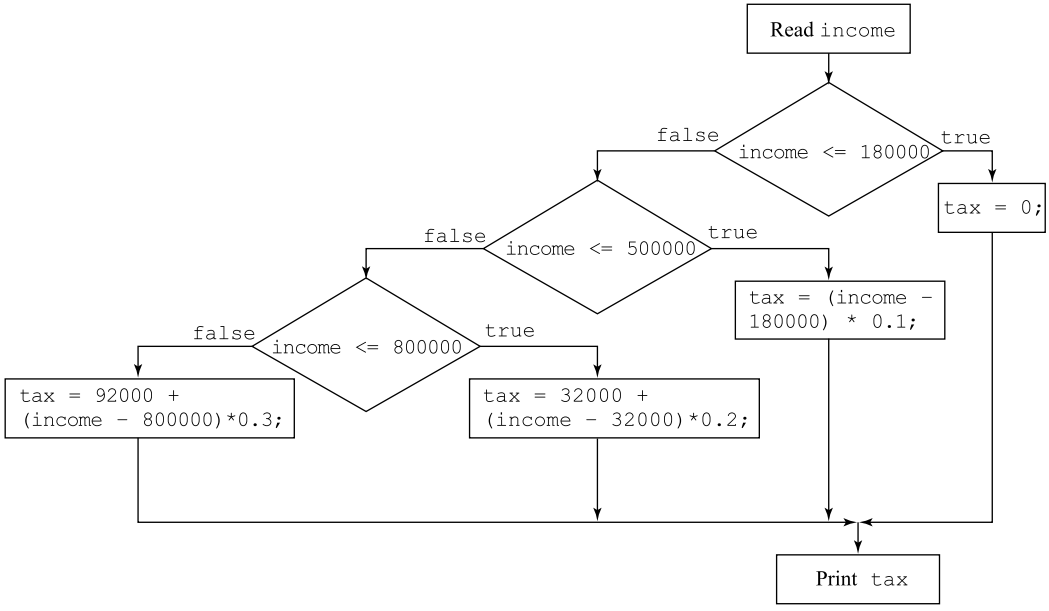


Fig. 6.5 *Flowchart for second income-tax program*

180000. But then, we don't need to check this again in the "new second if". So it suffices to just check if income is at most 50000. The third `if` statement also simplifies similarly.

Further, note that the original program would check each of its four conditions no matter which one is true, whereas in this program as soon as the first true condition is found, the corresponding consequent action is performed, and the subsequent conditions are not checked. Thus, the new program is more efficient than the previous program. Figure 6.5 shows the flowchart for the new program. By comparing to Figure 6.2, perhaps it is easier to appreciate how much different the new program is.

6.4 A DIFFERENT TURTLE CONTROLLER

The turtle-driving programs we saw in Chapter 1 required us to put information about the figure we wanted to draw right into the program, i.e., the exact sequence of forward and turn commands that we want to execute had to be written out in the program. We will now write a program which will allow the user to control the turtle during *during execution*.

Let us decide that the user must type the character 'f' to make the turtle go forward by 100 pixels, the character 'r' to make the turtle turn right by 90 degrees, and the character 'l' to make the turtle turn left by 90 degrees. Our program must receive these characters that the user types, and then move the turtle accordingly. Here it is.

```
main_program{
  char command;
  turtleSim();

  repeat(100){
    cin >> command;
    if (command == 'f') forward(100);
    else if (command == 'r') right(90);
    else if (command == 'l') left(90);
    else cout << "Not a proper command, " << command << endl;
  }
}
```

Remember that `char` data is really numerical, so it is perfectly acceptable to compare it using the operator ==. This program will execute 100 user commands to move the turtle before stopping. Try it!

6.4.1 "Buttons" on the Canvas

We can build "buttons" on the canvas using the `Rectangle` shapes of Section 5.3. We can control the turtle by clicking on the buttons. This gives yet another turtle controller.

```
main_program{
  initCanvas();

  const double bFx=150,bFy=100, bLx=400,bLy=100, bWidth=150,
  bHeight=50;
  Rectangle buttonF(bFx,bFy,bWidth,bHeight), buttonL(bLx,bLy,
  bWidth,bHeight);
```

```
  Text tF(bFx,bFy,"Forward"), tL(bLx,bLy,"Left Turn");
  Turtle t;

  repeat(100){
    int clickPos = getClick();
    int cx = clickPos/65536;
    int cy = clickPos % 65536;

    if(bFx-bWidth/2<= cx && cx<= bFx+bWidth/2 &&
       bFy-bHeight/2 <= cy && cy <= bFy+bHeight/2) t.forward(100);

    if(bLx-bWidth/2<= cx && cx<= bLx+bWidth/2 &&
       bLy-bHeight/2 <= cy && cy <= bLy+bHeight/2) t.left(10);
  }
}
```

The program begins by drawing the rectangles on the screen. Notice that we have not given the coordinate information of the buttons by writing numbers directly, but first created the names `bFx,bFy` and so on having specific values and then used these names in the button creation. Using such names is convenient: if you want to adjust the layout of buttons later, you just need to change the value of some name. Without names, you would have needed to make changes in every place the number appeared. In the present case, if you want to change the width of the rectangles, you just need to assign a different value to `bWidth`, instead of worrying in which all places the width value needs to be changed.

Next, text is put in the rectangles. Then we go into a loop. Inside, we wait for the user to click. We check whether the click is inside either of the two rectangles. This is done in the two `if` statements in the loop. Each check has two parts: we must check if the x coordinate of the click is between the left edge of the rectangle and the right edge, i.e. the left edge coordinate must be smaller or equal, and the right edge coordinate must be larger or equal. And correspondingly we must check for the y coordinate as well.

This program will only allow 100 clicks; we see later how to make the loop indefinitely or stop if some condition is met.

6.5 THE `switch` STATEMENT

In the turtle-control program, there was a single variable, `command`, depending upon which we took different actions. A similar situation arises in many programs. So C++ provides the `switch` statement so that we can express our code succinctly. The general form of the `switch` statement is

```
switch (expression){
  case constant_1:
     group(1) of statements usually ending with ``break;''
  case constant_2:
     group(2) of statements usually ending with ``break;''
  ...
  default:
     default-group of statements
}
```

The portion consisting of `default:` and the group of statements following that is optional. The expression `expression` must be of type `int`. Further, each `constat_i` in above is required to be an integer constant.

The statement executes in the following manner. First, the `expression` is evaluated. If the value is identical to `constant_i` for some *i*, then we start executing `group(i)` statements. We execute `group(i)` statements, then `group(i+1)` statements and so on, including `default-group` statements, unless we encounter a `break;` statement. If we encounter a `break` then the execution of the `switch` is complete, i.e. we do not execute the statements following the `break` but directly go to the statement in the program following the `switch` statement. If the value of `expression` is different from any of the `constant` values mentioned, then the `default-group` of statements is executed.

If a certain `group(i)` does not end in a `break`, then the execution is said to "fall-through" to the next group. Fall-throughs are considered to be rare.

Using a `switch`, our turtle-control program can be written as follows.

```
main_program{
  char command;
  turtleSim();

  repeat(100){
    cin >> command;
    switch(command){
      case 'f': forward(100);
                break;
      case 'r': right(90);
                break;
      case 'l': left(90);
                break;
      default: cout << "Not a proper command, " << command << endl;
    }
  }
}
```

As you can see, the new program is nicer to read.

Here is an example which has fall-throughs. Suppose we want to print the number of days in the nth month of the year, taking n as the input. Here is the program.

```
main_program{
  int month;
  cin >> month;
  switch(month){
    case 1:  // January
    case 3:  // March
    case 5:  // May
    case 7:  // July
    case 8:  // August
    case 10: // October
```

```
    case 12: // December
             cout << "This month has 31 days.\n";
             break;
    case 2:  // February
             cout << "This month has 28 or 29 days.\n";
             break;
    case 4:  // April
    case 6:  // June
    case 9:  // September
    case 11: // November
             cout << "This month has 30 days.\n";
             break;
    default: cout << "Invalid input.\n";
  }
}
```

Suppose the input is 5. Then the execution will start after the point labelled `case 5:`. It will fall through the cases 5,7,8,10 to case 12. In this, the number of days will be printed to be 31, and then a `break` is encountered. This will complete the execution of the `switch`.

The `switch` statement is considered somewhat error-prone because you may forget to write `break;`. So be careful.

6.6 CONDITIONAL EXPRESSIONS

C++ has a notion of a *conditional expression*, having the following form.

```
condition ? consequent-expression : alternate-expression
```

The evaluation of this proceeds as follows. First, the `condition` is evaluated. If it is `true`, then the `consequent-expression` expression is evaluated, and that is the value of the overall expression. The `alternate-expression` is ignored. If on the other hand the `condition` evaluates to false, then the `consequent-expression` is ignored, the `alternate-expression` is evaluated and the resulting value is the value of the overall expression.

Here are some simple examples.

```
int marks; cin >> marks;
int actualmarks = (marks > 100) ? 100 : marks;
char grade = (marks >= 35) ? 'p' : 'f';
```

In this, if `marks` read in were more than 100, then `actualmarks` would be capped to 100, else `actualmarks` would be set equal to `marks`. Further, if the marks are at least 35, then `grade` is set to 'p' (pass), otherwise to 'f' (fail).

Conditional expressions can be nested, i.e. the consequent or alternate expressions can themselves conditional expressions. This allows us to write a very compact but unreadable tax-calculation program.

```
main_program{
  float income; cin >> income;
```

```
  cout << (                                  //  **
           income <= 180000 ? 0 :
           income <= 500000 ? (income - 180000) * 0.1 :
           income <= 800000 ? 32000 + (income - 500000) * 0.2 :
           92000 + (income - 800000) * 0.3
          )                                  //  **
       << endl;
}
```

We merely read in the `income`, and then calculate the tax as an expression and directly print it out without storing it into a variable. In the above, the parentheses marked `**` are necessary. This is because the operator `<<` has higher precedence than the operator <=, i.e. by default C++ attempts to execute `<<` before <=.

The above program is very compact, but not recommended. Most programmers would consider it unreadable. However, the conditional expression without nesting is considered to be a useful construct.

6.7 LOGICAL DATA

An important part of the `if` statement are the conditions. We have already seen that a condition is either true or false, i.e. we can associate the value `true` or the value `false` with each condition. We have also seen that conditions can be combined in different ways. The resulting combination will also be `true` or `false`. We have also seen that there may be several equivalent ways of writing the same condition (as we saw for the second `if` statement of our first program). In this sense, conditions are similar to numerical expressions, numerical expressions have a value, numerical expressions can be combined to build bigger numerical expressions, we can have numerical expressions that are equivalent. In that case, why not treat conditions, as just another kind of data? This turns out to be a very good idea, and an algebra for manipulating conditions, or what we will hereafter refer to as logical expressions was developed by George Boole in 1940. C++ supports the manipulation and storage of logical data, and in honour of Boole, the data type for storing logical data is named `bool`. You have already seen this data type in Chapter 3, now we will do more interesting things with it.

First, we note that we can assign values of logical expressions to `bool` variables. Consider the following code.

```
float income; cin >> income;
bool lowIncome, midIncome, highIncome;
lowIncome = (income <= 180000);
midIncome = (income > 180000) && (income <= 800000);
highIncome = (income > 800000);
```

Suppose during execution, the value 200000 is given for `income`. Then after the execution of the subsequent statements, the variables `lowIncome, midIncome, highIncome` would respectively have the values `false, true, false`.

As you can see, the right-hand sides of the above assignment statements are conditions, and whatever the values these conditions have will been put in the corresponding left hand side variables.

As another example, let us define a `bool` variable that will be `true` if a character read from `cin` happens to be a lower case character. Note that this will happen if the ASCII value of the character is at least 'a' and at most 'z'. Thus, the code for this could be

```
char in_ch;
bool lowerCase;
cin >> in_ch;
lowerCase = (in_ch >= 'a') && (in_ch <= 'z');
```

We will next consider a more complex program which determines whether a given number `num` is prime or composite. The ability to store logical values will be useful in this program. To understand that program we will need to reason about expressions containing logical data. So we first discuss this.

6.7.1 Reasoning About Logical Data

As we discussed earlier, the same condition can be expressed in many ways. It is important to understand which expressions are equivalent.

First, let us make a few simple observations. For any logical value `v`, we have that `v || false` has the same value as `v`. The easiest way to check this is to try out all possibilities: if `v` is true, then `true || false` is clearly `true`. If `v` is `false`, then `false || false` is clearly `false`. Thus `false` plays the same role with respect to `||` that 0 plays with respect to numerical addition. More formally, `false` is said to be the identity for `||`. Likewise, `true && v` has the value `v` for any `v`. Or in other words, `true` is the identity for `&&`.

Another rule is the so called distributivity of `&&` over `||`. Thus, if `x, y, z` are boolean variables (or equivalently, conditions), then `(x && y) || z` is the same as `(x && z) || (y && z)`. In a similar manner, it turns out that `||` also distributes over `&&`.

Another important rule is that `x || !x` is always true, and hence we can replace such expressions with `true`. Similarly, `x && !x` can be replaced with `false`.

Finally, an important rule is De Morgan's Law. This says that `!x && !y` is the same as `!(x || y)`. Similarly `!x || !y` is the same as `!(x && y)`.

Consider first a condition such as `income <= 180000`. Income being at most 180000 is the same as it not being bigger than 180000. Hence we can write this condition also as `!(income > 180000)`.

While it is fine to be able to intuitively understand that the conditions

```
(income >  180000) && (income <= 500000)
```

and

```
!((income <=  180000) || (income > 500000))
```

are the same, you should also be able to deduce this given the rules given in this section.

6.7.2 Printing `Bool` Data

By default, `bool` data is printed numerically, with 0 being printed instead of `false` and 1 instead of `true`. However, see Appendix D.3.

You should also parenthesize conditional expressions while printing them. Thus, you should write `cout << (income <= 180000);` instead of `cout << income <= 180000;`. This is because `<<` which is also an operator, has a higher precedence than `<=` or other relational operators.

6.7.3 Determining Whether a Number is Prime

Determining whether a number is prime is an important problem, for which very sophisticated, very fast algorithms are known. We will only consider the simplest (and hence substantially slower than the fastest known) algorithms in this book.

Here is the most obvious idea. We go by the definition. A number n is prime if it has no divisors other than itself and 1. So it should suffice to check whether any number i between 1 and itself (both exclusive) divides it. If we find such an i then we declare x to be composite; otherwise it is prime.

This requires us to generate all numbers between 2 and $x-1$ (both inclusive this time) so that we can check whether they divide x. This is really the sequence generation pattern (Section 3.4.1) which we saw, say in the spiral drawing program of Section 3.4. There we made `i` take 10 values starting at 1. Now we want `i` to take the $x-2$ values from 2 to $x-1$. So here is the code fragment we should use:

```
i=2;
repeat(x-2){
  /*
  Here i takes values from 2 to x-1.
  */
  i = i + 1;
}
```

In each iteration of the loop, we can check whether `i` divides `x`. This is really the condition `(x % i) == 0`. We want to know whether any such condition succeeds. But this is nothing but a logical or, of the conditions that arise in each iteration. In other words, this itself is the accumulator pattern mentioned in Section 3.4.1. But we know how to implement that! We saw how to do it to calculate the sum in the average computing program of Section 3.4. We must maintain an accumulator variable which we set to the identity for the operator in question, and we update it in each step. Say we name our accumulator variable `factorFound` (since it will indicate whether a factor is found). Then we initialize it to `false`, the identity for the OR operation. Then in each step of the loop, we merely update `factorFound`, exactly as we updated `sum` in the average computation program. So our code fragment becomes

```
i=2;
factorFound = false;
repeat(x-2){
  factorFound = factorFound || (x % i) == 0;
  i = i+1;
}
```

At the end of this, `factorFound` will indeed be true if any of the expressions `(x % i) == 0` was `true`, for any value of `i`. Thus, following this code we simply print prime/composite depending upon whether `factorFound` is `false`/`true`. And at the beginning, we need to read in `x`, etc. The complete program is as follows.

```
main_program{  //Decide if x is prime.
  int x; cin >> x;
```

```
  int i=2;
  bool factorFound = false;
  repeat(x-2){
    factorFound = factorFound || (x % i) == 0;
    i = i+1;
  }

  if (factorFound) cout << x << " is composite." << endl;
  else cout << x << " is prime." << endl;
}
```

This program will be improved in several ways later. Once we find a factor of `x`, i.e. if in some iteration `x % i == 0` becomes true, we will set `factorFound` to `true`, and no matter what we do later, it cannot become false. So why even do the remaining iterations? This is indeed correct: if we are testing if 102 is prime, we will discover in the first iteration itself that 102 is divisible by 2, i.e. 2 is a factor and that 102 is composite. So we should prematurely stop the loop and not do the remaining iterations. In the next chapter we will see how this can be done.

Note by the way that effect of `factorFound = factorFound || (x % i) == 0;` can also be had by writing `if(x % i == 0) factorFound = true;` This doesn't look like accumulation, but has the same effect.

6.8 PITFALLS

There is a potential pitfall associated with the use of the operators = and ==. In mathematics, the operator = is used to denote comparison, and since most of us learn mathematics before programming, we are likely predisposed to use = to mean comparison even in C++, rather than ==. This will lead to errors. The situation is more serious than what you might think at first glance. If you write code such as

```
if(p = 25) q = 37;
```

when you mean `if(p == 25) q = 37;` the compiler will not regard it as an error. This is because assignment is also an expression, and it evaluates to the value that got assigned. Thus in this case, `p = 25`, the value is 25. Further, the compiler will, on its own, try to convert this value to a boolean value. For this the rule of conversion is a bit non-intuitive: any non-zero value becomes `true` and only 0 becomes `false`. Thus in the execution of the above statement, the assignment `q = 37` will always happen.

Many compilers can be asked to warn if they encounter such statements which most likely are silly mistakes made by the programmer. Indeed, the GNU C++ compiler will give a warning if it sees such statements in your program, provided you invoke it using the option `-Wparentheses`. And in fact, `s++` which you use with `simplecpp` indeed calls the GNU C++ compiler with this option, so you will get these warnings already if you compile with `s++`. If you really intended the statement to mean the assignment expression (and did not mistakenly write = instead of ==), then you can merely put the expression inside a pair of parentheses and write `if((p = 25)) q = 37;`. This effectively declares your firm intent that you mean `p = 25` to be an assignment expression. Thus, in this case, no warning will be issued even if you use the option `-Wparentheses`.

Another pitfall concerns nesting of `if` statements, say if the `consequent` of an `if` is itself another `if` statement.

```
if(a > 0) if(b > 0) c = 5; else c = 6;
```

This is treated by the compiler to mean

```
if(a > 0) {if(b > 0) c = 5; else c = 6;}
```

In other words, the `else` joins with the innermost `if`, and the outer `if` is left without an `else` clause. Keeping track of such rules is rather cumbersome, so it is best if you insert the braces yourself. Of course, if you meant to associate the else with the outer `if` you could have written

```
if(a > 0) {if(b > 0) c = 5;} else c = 6;
```

If you omit the braces, then the compiler again will warn you if you have used the `-Wparentheses` option. Note that the compiler will have compiled your program as per the rules of C++, even when it issues a warning. However, you should treat compiler warnings as suggestions to improve the readability of your code. Indeed, if you use parentheses or braces as suggested above, you make your code more readable to other programmers as well.

We have already noted that the `switch` statement is somewhat error-prone, because of the possibility of forgetting to use the `break` statement.

A slightly uncommon pitfall should also be mentioned. If you attempt to compare signed and unsigned `int`s, the results can be unexpected. This is because C++ simply treats the bit patterns representing both (Section 2.4) as unsigned integers and compares them. This may cause a negative number stored in a signed `int` to be declared larger than a positive number stored in an unsigned `int`. Try the following:

```
int i = -1;
unsigned int j=100;
cout << (i < j) << endl;
```

You will see that the comparison comes out false, i.e. `i` is considered bigger. Some C++ compilers may alert you by issuing a warning when they encounter code that compares a signed `int` an unsigned `int`. You may choose to ignore the warning, say if you know that the signed `int` value will also be positive. But it is better if you write your code as

```
int i = -1;
unsigned int j=100;
cout << (i < int(j)) << endl;
```

This way there will not be a warning message from the compiler.

6.9 CONCLUDING REMARKS

In this chapter, we saw how we can ask the computer to conditionally execute some statement or group of statements, using the `if` and `switch` statements, with the `if` itself coming in three forms. As you might realize, we don't really *need* the additional two forms of the `if` statement, nor the `switch` statement. But by using the other forms of the `if` we can avoid making too many comparisons. Also, using the `switch`, we can highlight that different actions need to be performed depending upon the

values of a single expression. Thus all the statements are useful, and it is important to understand them well.

We also studied conditional expressions.

We studied the notion of logical variables, which can be used to store the results of comparisons so that they can be used later if needed, without having to perform comparisons again. More than that a logical variable can be used to *accumulate* the results of comparisons performed inside a loop. We saw this come in handy in the program to determine if a number is prime.

We also noted some pitfalls related to `if` and `switch`.

EXERCISES

1. Modify the turtle program so that the user can specify how many pixels the turtle should move, and also by what angle to turn. Thus, if the user types "f100 r90 f100 r90 f100 r90 f100" it should draw a square.
2. Write a program that reads 3 numbers and prints them in non-decreasing order.
3. Write a program that reads in 4 numbers and prints them in non-decreasing order. Argue that your answer is correct.
4. Write a program which takes as input a number denoting the `year`, and says whether the year is a leap year or not a leap year.
5. Write a program that takes as input a number `y` denoting the year and a number `d`, and prints the date which is the `d`th day of the year `y`. Suppose `y` is given as 2011 and `d` as 62, then your program should print "3/3/2011".
6. Write a program that takes as input 3 numbers a, b, c and prints out the roots of the quadratic equation $ax^2 + bx + c = 0$. Make sure that you handle all possible values of a, b, c without running into a division by zero or having to take the square root of a negative number. Even if the roots are complex, you should print them out suitably.
7. Suppose we wish to write a program that plays cards. The first step in such a program would be to represent cards using numbers. In a standard deck, there are 52 cards, 13 of each suite. There are 4 suites: spades, hearts, diamonds, and clubs. The 13 cards of each suit have the denomination 2,3,4,5,6,7,8,9,10,J,Q,K,A, where the last 4 respectively are short for jack, queen, king and ace. It is natural to assign the numbers 3,2,1,0 to the suites respectively. The denominations 2–10 are assigned numbers same as the denomination, whereas the jack, queen, king, and ace are respectively assigned the numbers 11, 12, 13, and 1 respectively. The number assigned to a card of suite s and denomination d is then $13s + d$. Thus, the club ace has the smallest denomination, 1, and the spade king the highest, 52. Write a program which takes a number and prints out what card it is. So given 20, your program should print "7 of diamonds", or given 51, it should print "queen of spades".
8. Write a program that takes a character as input and prints 1 if it is a vowel and 0 otherwise.
9. Can you write the program to determine if a number is prime without using a `bool` variable? *Hint*: Count how many factors the number has.

10. A number is said to be perfect if it is equal to the sum of all numbers which are its factors (excluding itself). So, for example, 6 is perfect, because it is the sum of its factors 1, 2, 3. Write a program which determines if a number is perfect. It should also print its factors.

11. Write a program which prints all the prime numbers smaller than n, where n is to be read from the keyboard.

12. Write a program that reads in three characters. If the three characters consist of two digits with a '.' between them, then your program should print the square of the decimal number represented by the characters. Otherwise your program should print a message saying that the input given is invalid.

13. Make an animation of a ball bouncing inside a rectangular box. Assume that the box is attached to the ground, and the ball moves horizontally inside, without friction. Further assume for simplicity that the ball has an elastic collision with the walls of the box, i.e. the velocity of the ball parallel to the wall does not change, but the velocity perpendicular to the wall gets negated. Put the pen of the ball down so that it traces its path as it moves. You can either read the ball position and velocity from the keyboard, or you can take it from clicks on the canvas. Move the ball slowly along its path so that the animation looks nice. The animation should run till the ball collides with some wall n times, where n is read from the keyboard.

14. Modify the animation assuming that the box has mass equal to the ball, and is free to move in the x direction, say it is mounted on frictionless rails parallel to the x direction. Note that now in each collision the velocity of the box will also change. If the box has velocity v and the ball has velocity u parallel to the x axis at the time of the collision, then these velocities will be exchanged during collision, i.e. will become u and v respectively. Show the animation of this system. You may want to start off the system with the x component of the ball velocity equalling the negative of the velocity of box. This will ensure that the box will not move out of the screen.

15. * In the hardest version of the ball-in-a-box problem, the box is sitting on a frictionless surface, and is free to turn. Now after a collision, the box will in general start rotating as well as translating. Assume for simplicity that the mass of the box is uniformly distributed along its 4 edges, i.e. the base is massless.

16. At the top of Figure 6.6 are shown pictorial representations of a NOT gate, an AND gate and an OR gate. The lines emanating to the left from each gate are its *input*s. The line emanating to the right is the *output*. Gates can be used to build digital circuits, one such circuit is shown in the lower portion of the figure. The devices inside an AND gate are such that the output takes the value 1 if both inputs have the value 1. If any input is a 0, then the output is a 0. The output of an OR gate is 0 if both inputs are 0, and it is 1 if even one of the inputs is 1. The output of a NOT gate is 1 if the input is 0, and 0 if the input is 1.

The value appearing at the output of a gate can be fed to an input of some gate by connecting a wire between the two, shown by a line in the figure. The wires marked a, b bring values from outside the circuit. Some wires in the figure are seen to cross—these are not to be considered electrically connected unless a solid dot is present at the intersection. By knowing the values on a, b you can determine the values on the outputs of the NOT gates, and the output of the lowest AND gate. By knowing the value of a and the values of the outputs of the NOT gate you should be able to determine the value at the output of the upper AND gates. Finally, using the values of the outputs of the upper AND gates, you should be able to determine value at the output of the

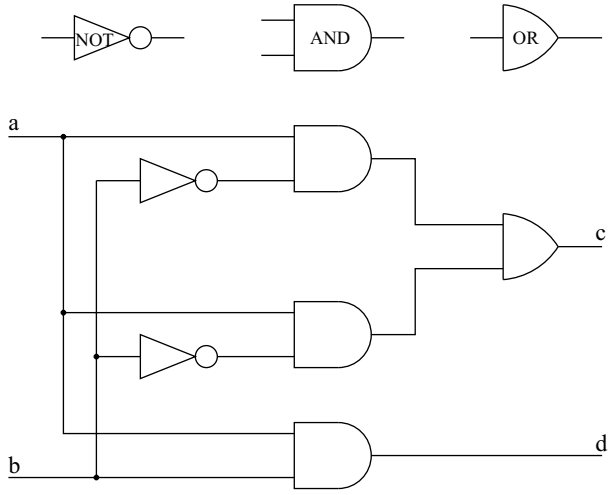


Fig. 6.6 *Circuit components and a circuit*

OR gates. Thus, you should be able to determine the values on wires c, d knowing the values on wires a, b.

Write a program that reads the value on a, b and prints the values on c, d.

17. Develop a mini drawing program as follows. The screen should have buttons called "Line" and "Circle" which a user can click to draw a line or a circle. After a user clicks on "Line", you should take the next two clicks to mean the endpoints of the line, and so after that a line should be drawn connecting those points. For now, you will have to `imprint` that line on the canvas. Similarly, after clicking "Circle", the next point should be taken as the center, and the next point as a point on the circumference. You can also have buttons for colours, which can be used to select the colour before clicking on "Line" or "Circle".

CHAPTER 7

Loops

Consider the following *mark-averaging* problem:

> From the keyboard, read in a sequence of numbers, each denoting the marks obtained by students in a class. The marks are known to be integers in the range 0 to 100. The number of students is not told explicitly. If any negative number is entered, it is not to be considered the marks of any student, but merely a signal that all the marks have been entered. Upon reading a negative number, the program should print the average mark obtained by the students and stop.

Using the statements you have learned so far, there is no nice way in which the above program can be written. It might seem that the program requires us to do something repeatedly, but the number of repetitions equals the number of students, and we don't know that before starting on the repetitions. So we cannot use the `repeat` statement, in which the number of times to repeat must be specified before the execution of the statement starts.

In this chapter, we will learn the `while` loop statement which will allow us to write the program described above. We will also learn the `for` loop statement, which is a generalized version of the `while` statement. All the programs you have written earlier using the `repeat` statement can be written using `while` and `for` instead, and often more clearly. The `repeat` statement is not really a part of C++, but something we added through the package Simplecpp because we didn't want to confuse you with `while` and `for` in the very first chapter. But having understood these more complex statements you will find no real need for the `repeat` statement. So we will discontinue its use from the next chapter.

7.1 THE `while` STATEMENT

The most common form of the `while` statement is as follows.

```
while (condition) body
```

where `condition` is a boolean expression, and `body` is a statement, including a block statement. The `while` statement executes as follows.

1. The `condition` is evaluated.

2. If the condition is `false`, sometimes described as "if the condition fails", then the execution of the statement is complete without doing anything more. Then we move on to execute the statement following the `while` statement in the program.
3. If the `condition` is `true`, then the `body` is executed.
4. Then we start again from step 1 above.

This is shown as a flowchart in Figure 7.1.

Each execution of the `body` is called an *iteration*, just as it was for the `repeat`. Each iteration might change the values of some of the variables so that eventually `condition` will become `false`. When this happens, it will be detected in the subsequent execution of step 1, and then step 2 will cause the execution of the statement to terminate.

As you can perhaps already see, this statement is useful for our mark-averaging problem. But before we look at that let us take some simpler examples.

First, we note that using a `while`, it is possible to do anything that is possible using a `repeat`. To illustrate this, here is a program to print out a table of the cubes of numbers from 1 to 100. Clearly, you can also write this using `repeat`.

```
main_program{
  int i=1;
  while(i <= 100){
    cout << "The cube of " << i << " is " << i*i*i << endl;
    i = i + 1;
  }
  cout << "Done!" << endl;
}
```

The execution will start by setting `i` to 1. Then we check whether `i` is smaller than or equal to 100. Since it is, we enter the body. The first statement in the body causes us to print "The cube of 1 is 1",

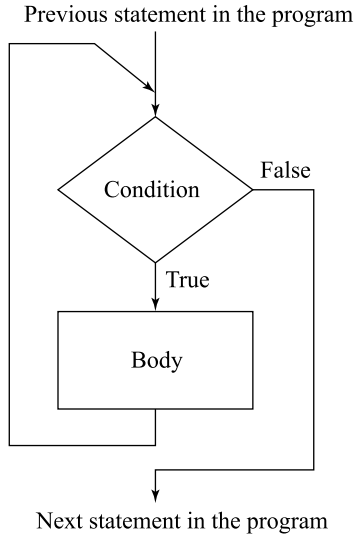


Fig. 7.1 *While statement execution*

because i has value 1. Then we increment i. After that we go back to the top of the statement, and check the condition again. Again, we discover that the current value of i, 2, is smaller than or equal to 100. So we print again, this time with i=2, so what gets printed is "The cube of 2 is 8". We again execute the statement i = i + 1;, causing i to become 3. We then go back and repeat everything from the condition check. In this way it continues, until i is no longer smaller than or equal to 100. In other words, we execute iterations of the loop until (and including) i becomes 100. When i becomes 101, the condition i >= 100 fails, and so we go to the statement following the loop. Thus we print "Done!" and stop. But before this, we have executed the loop body for all values of i from 1 to 100. Thus, we will have printed the cube of all the numbers from 1 to 100.

7.1.1 Counting the Number of Digits

We consider a more interesting problem: read in a non-negative integer from the keyboard and print the number of digits in it. The number of digits in a number n is simply the smallest positive integer d such that $10^d > n$. So our program could merely start at $d = 1$, and try out successive values of d until we get to a d such that $10^d > n$.

Thus, we have to generate the sequence $10, 10^2, 10^3, \ldots$; but this is just the sequence-generation idiom. We should stop generating the sequence as soon as we generate a sequence element, say 10^d which is larger than n. In other words, we should not stop while $10^d \leq n$. This is what the following code does.

```
main_program{
  int n;  cout << "Type a number: ";  cin >> n;

  int d = 1, ten_power_d=10;
           // ten_power_d will always be set to 10 raised to d

  while(n >= ten_power_d){  // if loop entered,
                            // number of digits in n must be > d
    d++;                    // so we try next choice for d
    ten_power_d *= 10;
  }

  cout << "The number has " << d << " digits." << endl;
}
```

Let us see what happens when we run the program. Say in response to the request to type in a number, we entered 27. Then we would set d to 1 and ten_power_d to 10. Then we would come to the while loop. We would find that n, which equals 27 is indeed bigger than or equal to ten_power_d) which equals 10. So we enter the loop. Inside the loop, we add 1 to d so that it becomes 2, and we multiply ten_power_d by 10, so it becomes 100. We then go back to the beginning of the loop and check the condition. This time we would find that n whose value is 27 is smaller than ten_power_d whose values is 100. So we do not enter the loop but instead go to the statement following the loop. Thus we would print the current value of d, which is 2, as the number of digits. This is the correct answer: the number of digits in 27 is indeed 2.

7.1.2 Mark Averaging

This problem, like many problems you will see later, is what we might call a *data streaming* problem. By that we mean that the computer receives a stream (sequence) of values, and we are expected to produce something when the stream terminates. Occasionally, we may be expected to print out a stream of values as well, but in the current problem, we have to only print out their average. A general strategy for tackling such problems is to ask yourself: what information do I need to remember at a point in execution when some n values of the stream have been read? The answer to this often suggests what variables are needed, and how they should be updated.

For the mark-averaging problem, we know what we want at the end: we want to print out the average. To calculate the average we need to know the sum of all the values that we read, and a count of how many values we read. So at an intermediate point in the program, when some n values have been read, we should keep track of n as well as their sum. We don't need to remember the individual values that we have read so far! So it would seem that we should keep a variable `sum` in which we maintain the sum of the values that we have read till any point in time. We should also maintain a variable `count` which should contain the number of values we read. Both variables should start off 0. We will have a repeated portion in which we read a value, and for this we will have a variable called `nextmark`. Using these it would seem that we need to do the following steps repeatedly.

1. Read a value into `nextmark`.
2. If `nextmark` is negative, then we have finished reading, and so we go on to calculating and printing the average.
3. If `nextmark` is non-negative, then we add `nextmark` to `sum`, and also increment `count`.
4. We repeat the whole process from step 1.

In this, we have not written down the process of calculating the average, etc. But that is simply dividing `sum` by `count`. Figure 7.2(a) shows this as a flowchart.

Can we express this flowchart using the `while` statement? For this, you would need to match the pattern of the flowcharts of Figure 7.1 and Figure 7.2(a). It seems natural to match the `condition` in the former with the test `nextmark >= 0` in the latter. But there is an important difference in the structure of the two flowcharts. In Figure 7.1, the condition test is the first statement of each iteration, while in Figure 7.2(a), the first statement is reading the data, and only the second statement is the condition check.

The crucial question then is: can we somehow modify the flowchart of Figure 7.2(a) so that the execution remains the same, but the new flowchart matches the pattern of Figure 7.1? Suppose we decide to move the box labelled A upwards above the point P where two branches merge. We do not want to change what happens on each branch that enters P, so then it simply means that we must place a copy of A on both branches coming into P. This gives us the flowchart of Figure 7.2(b). As you can see, the two flowcharts are equivalent in that they will cause the same statements to be executed no matter what input is supplied from the keyboard.

Note now that box B and the left copy of A in Figure 7.2(b) are executed successively, so we can even merge them into a single box containing 3 statements. This new box can become the body of a `while` statement, and box C the `condition`. Thus, we can write our code as follows.

```
main_program{
  float nextmark, sum=0;
  int count=0;
```

```
  cin >> nextmark;          // right copy of box A

  while(nextmark >= 0){     // box C
    sum = sum + nextmark;   // Box B
    count = count + 1;      // Box B

    cin >> nextmark;        // left copy of box A
  }

  cout << "The average is: " << sum/count << endl;
}
```

The above program assumes that there will be at least one true mark, so that count will not be zero at the end.

Note the general idea carefully: the natural way of expressing our program could involve a test in the middle of the code we wish to repeat. In such cases, we can get the test to be at the top by moving around some code and also making a copy of it. Soon you will start doing this automatically.

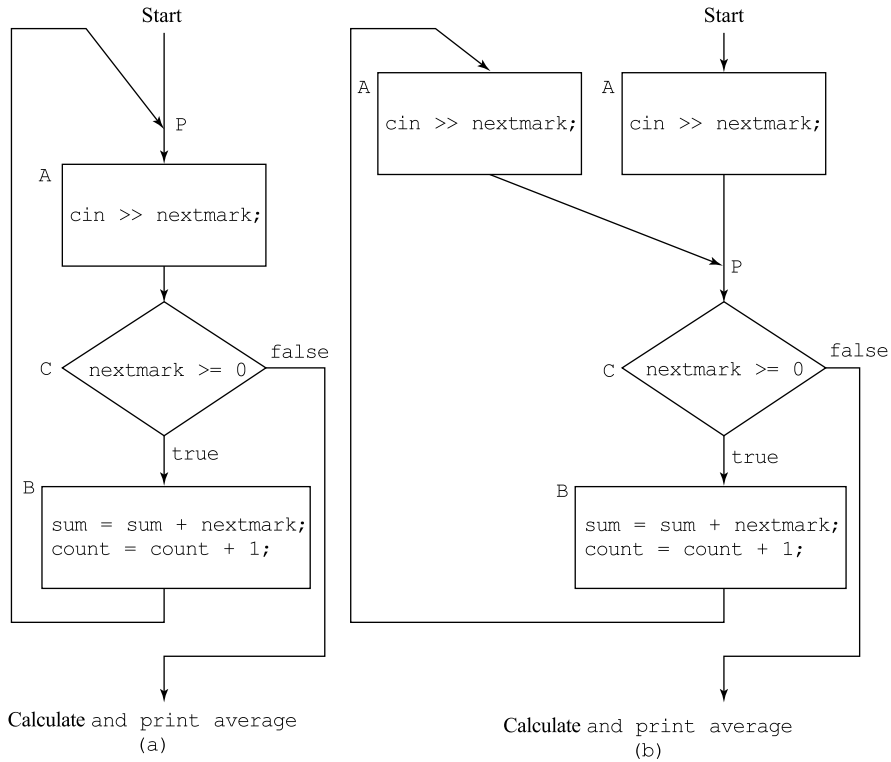


Fig. 7.2 *Flowcharts for averaging*

7.2 THE break STATEMENT

C++ allows a `break;` statement to be used inside the `body` of a `while` (both forms). The `break` statement causes the execution of the containing `while` statement to terminate immediately. If this happens, execution is said to have *broken* out of the loop. Here is a different way of writing our mark-averaging program using the `break` statement:

```
float nextmark, sum=0;
int count=0;

while(true){
  cin >> nextmark;
  if(nextmark < 0) break;
  sum = sum + nextmark;
  count = count + 1;
}
cout << sum/count << endl;
```

The first point to note here is that `condition` is given as `true`. This means that the statement will potentially never terminate! However, the statement does terminate because of the `break` statement in the `body`. After the `nextmark` is read, we check if it is negative – if so the statement terminates immediately, and we exit the loop. If the `nextmark` is non-negative, we add `nextmark` to `sum` and so on. The result of this execution will be the same as before. Note that this is similar to the flowchart of Figure 7.2(a).

Is the new program better than the old one? It is better in one sense: the statement `cin » nextmark;` is written only once. In general, it is a good idea to not duplicate code. First, this keeps the program small, but more importantly it prevents possible errors that might arise later. For example, suppose you later decide that just as you read `mark` you also want to print what was read. If the reading code is in several places, then you might forget to make the change in all the places. Another question is: which program *feels* more natural? You may probably consider the program with the `break` more natural—as we remarked, its structure matches the first flowchart we drew. Naturalness is indeed an important criterion in deciding what is better.

The old code was better in that the `condition` for terminating the loop was given up front, at the top. In the new code, the reader needs to search a little to see why the loop will not execute *ad infinitum*. This could be cumbersome if the loop body was large. So we cannot unequivocally say that the new code is better.

Note finally that in case of nested loops, the `break` statement allows us to break out of only the innermost loop statement in which it is contained.

7.3 THE continue STATEMENT

What if someone typed in a number larger than 100 for `nextmark`? Since we are assuming that marks are at most 100, we could perhaps ignore the numbers above 100 as being erroneous. This is conveniently expressed using the `continue` statement.

When a `continue` statement is encountered during execution, the remaining part of the loop body is ignored. The control goes to the top of the loop, and checks the `condition` and begins the next iteration if check comes out `true`, and so on.

The main loop in the program can be written as follows using the `continue` statement.

```
while(true){
  cin >> nextmark;
  if(nextmark > 100){
    cout << "Larger than 100, ignoring." << endl;
    continue;
  }
  if(nextmark < 0) break;
  sum = sum + nextmark;
  count = count + 1;
}
```

If `nextmark` is bigger than 100, then the message is first printed, and then the rest of the loop body is skipped. The next iteration is begun, starting with the condition check, which in this case is always `true`.

Note finally that in case of nested loops, the `continue` statement causes execution to skip the rest of the `body` of the innermost loop statement containing it.

7.4 THE `do while` STATEMENT

The while statement has a variation in which the `condition` is tested at the end of the iteration rather than at the beginning. It is written slightly differently. The form is

```
do body while (condition);
```

This is executed as follows.

1. The `body` is executed.
2. The `condition` is evaluated. If it evaluates to `true`, then we begin again from step 1. If the body evaluates to `false` then the execution of the statement ends.

In other words, in the `do-while` form, the body is executed at least once. You will observe that the `do-while` form above is equivalent to the following code using only the `while`:

```
body
while (condition) body
```

So you may wonder: why do we have this extra form as well? As you can see, the new form is more compact if you don't want the condition checked for the first iteration. Here is a typical example.

```
main_program{
  float x;
  char response;

  do{
    cout << "Type the number whose square root you want: ";
    cin >> x;
    cout << "The square root is: " << sqrt(x) << endl;
```

```
    cout << "Type y to repeat: ";
    cin >> response;
    }
  while(response == 'y');
}
```

This will keep printing square roots as long as you want.

7.5 THE `for` STATEMENT

Suppose you want to print a table of cubes of the integers from 1 to 100. You would solve this problem using the following piece of code.

```
int i = 1;
repeat(100){
  cout << i << `` `` << i*i*i << endl;
  i = i + 1;
}
```

The variable `i` plays a central role in this code. All iterations of the `repeat` are identical, except for the value of `i`. Further, `i` changes from one iteration of the loop to another in a very uniform manner, in the above case it is incremented by 1 at the end of each iteration. This general code pattern: that there is a certain variable which takes a different value in each iteration and the value determines how the iteration will execute, is very common. Because of this, the designers of C++ (and other programming languages) have provided a mechanism for expressing this pattern very compactly. This mechanism is the `for` statement. Using the `for` statement, we can print a table of cubes as follows.

```
for(int i=1; i <= 100; i = i + 1)
  cout << i << ` ` << i*i*i << endl;
```

To see how this works, let us consider the `for` statement in its general form:

```
for(initialization ; condition ; update) body
```

In this, `initialization` and `update` are required to be expressions, typically assignment expressions. As you might remember, an assignment expression is simply assignments to a variable without including the semicolon, e.g. `i = i + 1`. Further, we may include the definition along with the assignment e.g. `int i = 0`. As you might expect `condition` must be a boolean expression. The last part, `body` may be any C++ statement, including a block statement. In our example above, the `body` consisted of the statement `cout << i << " " << i*i*i << endl;`.

The execution of a `for` statement starts with the execution of `initialization`. Then `condition` is evaluated. If `condition` is false, then the statement terminates. If the `condition` is true, the statements in the `body` are executed followed by the `update`. We repeat this process again starting from evaluation of `condition`. This is shown as a flowchart in Figure 7.3.

Note that any of the fields `initialization`, `condition`, `update` or `body` can be empty. If the `condition` is empty, then it is taken as `true`.

The variable named in `initialization` and `update` is customarily called the *control variable* of the loop. Typically, `initialization` assigns an initial value to the control variable, and the

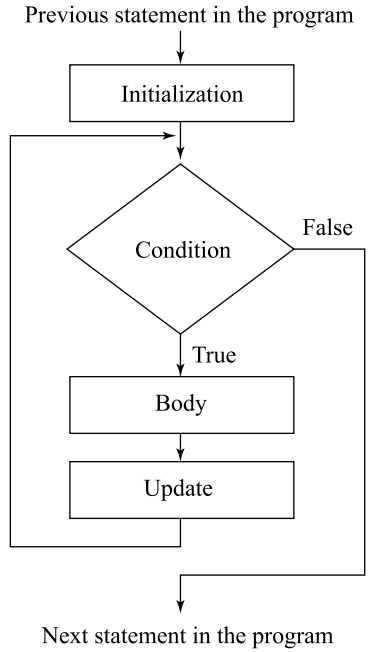


Fig. 7.3 *For statement execution*

`update` says how the variable must change from one iteration to the next. As you can see, in our cube table example, the `update` indeed adds 1 to the control variable.

You probably also see why the statement is called a `for` statement. It is because we execute the `body` many times, *for* different values of the control variable.

7.5.1 Variables Defined in `initialization`

As mentioned above, the `initialization` can contain a variable definition, as in our cube-table program. This variable is created during `initialization`, and is available throughout the execution of the `for` statement, i.e. during all the iterations. It is destroyed only when the execution of the `for` statement ends. Thus such a variable cannot referred to outside the `for`. If the value of the variable is useful after the `for` execution is over, then the variable should be defined before the `for` statement, and only initialized in `initialization`.

What if I define a variable `i` in `initialization`, but an `i` has already been defined earlier? So consider the following code.

```
int i=10;

for(int i=1; i<=100; i = i + 1) cout << i*i*i << endl;

cout << i << endl;
```

In this case, we will have shadowing, as discussed in Section 3.6.3. In particular, the `i` defined in the first statement will be different from the one defined in the `for` statement, but it will be the same as

the one in the last statement! Thus, the `for` statement will print a table of cubes as before. The last statement will print 10, because the variable `i` referred to in it is the variable defined before the `for`.

7.5.2 `Break` and `continue`

If a `break` statement is encountered during the execution of `body`, then the execution of the `for` statement finishes. This is exactly as in the `while` statement.

If the `continue` statement is encountered, then the execution of the current iteration is terminated, as in the `while` statement. However, before proceeding to the next iteration, the `update` is executed. After that control continues with the next iteration, starting with checking `condition`, and so on.

7.5.3 Style Issue

You may well ask: why should we learn a new statement if it is really not needed? Indeed, any program that uses a `for` statement can be rewritten using a `while`, with a few additional variables and assignments.

The reason concerns style. It is much the same as why we speak loudly on certain occasions and softly on others: our softness/loudness help the listener understand our intent in addition to our words. Likewise, when I write a `for` statement, it is very clear to the reader that I am using a certain common programming idiom in which there is a control variable which is initialized at the beginning and incremented at the end of each iteration. If I use either a `while` statement or a `repeat` statement, then the reader does not immediately see all this.

7.5.4 Determining if a Number is Prime

In Section 6.7.3, we developed a program to determine if a number is prime. We remarked there that the program can be made more efficient by noting that once we find a factor for the given number, we can stop checking for additional factors and immediately report that the number is composite. We can implement this idea using the `for` statement as follows.

To determine whether `x` is prime, our program of Section 6.7.3 checked if `x` is divisible by `i`, where `i` goes from 2 to `x-1`. Clearly, `i` can serve nicely as a control variable. Furthermore, once we detect that `x` is divisible by `i`, we can break out of the loop.

```
main_program{
  int x; cin >> x;

  bool found = false;
  for(int i=2; i < x; i++){// if the loop is entered we know that
                           // x is not divisible by 2...i-1
    if(x % i == 0){ found = true; break;}
  }

  if(found) cout << "Composite.\n";
  else cout << "Prime.\n";
}
```

Note the comment we have added to the code. It expresses how our knowledge about whether `x` is prime evolves as the program goes through the loop. No matter which iteration it is, whatever value `i` has, we know that `x` is not divisible by the numbers in the range 2 through `i-1`. Here, if `i` is 2, the

range is empty, so our claim is to be considered true. Comments such as this one are useful to people reading the code, they help in understanding what is going on.

7.6 UNCOMMON WAYS OF USING `for`

Most often, the `initialization` and `update` in the `for` statement each consists of an assignment to a single variable. However, there are other possibilities too, as we will see in this section.

7.6.1 Comma-separated Assignments

Here is how we might solve the digit-counting problem of Section 7.1.1 using the `for` statement.

```
main_program{
  int n; cin >> n;

  int d, ten_power_d;
  for(d=1, ten_power_d = 10; ten_power_d <= n; d++,
  ten_power_d *= 10){};

  cout << "The number has " << d << " digits." << endl;
}
```

There are two noteworthy features of the `for` statement in the above code. First, the `initialization` and `update` both consist of two assignments separated by a comma. This is allowed. The comma is considered to be an operator in such a context (Appendix C.2), and it merely joins together two assignments!

The second noteworthy aspect is that the above `for` statement has an empty `body`. This is acceptable. In fact, it is acceptable to even omit `{}`, but this is not recommended. By writing out `{}`, you make it clear to the reader that you have deliberately written an empty body rather than possibly have forgotten to supply one.

The above code is very compact, but might be considered tricky by some. The point of to note, of course, is that comma separated assignments can be used as `initialization` and `update` in a `for` statement in general.

7.6.2 Input in `initialization` and `update`

Here is how we could write the mark-averaging code using a `for` statement.

```
main_program{
  float nextmark,sum=0;
  float count=0;

  for(cin >> nextmark; nextmark >= 0; cin >> nextmark){
    count++;
    sum += nextmark;
  }
  cout << sum/count;
}
```

We said that `initialization` and `update` in a `for` statement must be expressions; but it turns out that `cin » nextmark` is an expression! We will discuss what value it returns in Section 13.6.3. But right now the value does not concern us; so you can go ahead and use such input expressions in `initialization` and `update`.

Some programmers may like this way of writing the program. It is a bit unconventional. However, it does make sense to consider `nextmark` to be a control variable for this program.

7.7 THE GREATEST COMMON DIVISOR

We will now discuss what is one of the most elegant, oldest, and useful algorithms ever: the algorithm for finding the Greatest Common Divisor (GCD) due to Euclid, from around 300 BC. As you know, the inputs for this problem are positive integers m, n. We are required to compute their GCD, which is defined to be the largest integer that divides m, n both. It can be written using a single `while` loop.

The algorithm for this, as taught in primary schools, is to factorize both the numbers, and then the greatest common divisor (GCD) is the product of the common factors. Another possibility is to go with the specification: examine the numbers between 2 and $\min(m, n)$, and find the largest one that divides both. This will work, but is slower than the primary school method.

Euclid's algorithm is much faster than both these methods. The starting point for it is a relatively simple observation:

> If d is a common divisor of positive integers m, n, then it is a common divisor also of $m - n, n$, assuming $m > n$.

The proof is simple: Since d divides m, n we have $m = pd, n = qd$, for integers p, q. Thus $m - n = (p - q)d$, and hence d divides $m - n$ also. By a similar argument you can also prove the converse, i.e. if d is a common divisor of $m - n, n$, then d is a common divisor of m, n also.

Thus, we have shown that every common divisor of m, n is also a common divisor of $m - n, n$, and vice versa. But then it means that the set of common divisors of m, n is identical to the set of common divisors of $m - n, n$. Thus, the greatest in the first set must be the greatest in the second set, i.e. $GCD(m, n) = GCD(m - n, n)$.

The last statement has profound consequences. It should be read as saying: if you want the GCD of m, n, you may instead find the GCD of $m - n, n$ assuming $m > n$. This could be considered progress, because intuitively, you would think that finding the GCD of smaller numbers should be easier than finding the GCD of larger numbers.

Let us take an example. Suppose we want to find the GCD of 3977, 943. Thus, we have $GCD(3977, 943) = GCD(3977 - 943, 943) = GCD(3034, 943)$. But there is no reason why we should use this idea just once: we can use it many times. Thus we get $GCD(3034, 943) = GCD(2091, 943) = GCD(1148, 943) = GCD(205, 943)$. At this point you might realize that we can subtract all multiples in one shot, and the result is simply the remainder when dividing the original number 3977 by 943. Thus, we could more directly have written $GCD(3977, 943) = GCD(3977\%943, 943) = GCD(205, 943)$.

Because GCD is a symmetric function we can subtract multiples of m just as well as n. Thus $GCD(205, 943) = GCD(205, 943\%205) = GCD(205, 123)$. This further simplifies: $GCD(205, 123) = GCD(205\%123, 123) = GCD(82, 123) = GCD(82, 123\%82) = GCD(82, 41)$. At this point, if we try to apply our rule we get $82\%41 = 0$, i.e. the smaller of

the numbers divides the larger, and so it must be the GCD. Thus we have obtained, overall, that GCD(3977,943)=41.

We can summarize the ideas above into a simple theorem.

Theorem 1 (Euclid) *Suppose m, n are positive integers. If $m\%n = 0$, then $GCD(m, n) = n$. Otherwise $GCD(m, n) = GCD(m\%n, n)$.*

This is enough to write a program. The program starts by reading the numbers into variables `m,n`. Then in each iteration, we will use Euclid's theorem to obtain new values for `m,n` such that the GCD of the new values is the same as the GCD of the old values. The new values will keep on getting smaller, but we know that this cannot happen indefinitely. Hence there must come a time when we cannot reduce the values of `m,n` using Euclid's theorem. But this can happen only when `n` divides `m`, whereupon we can print out `n` as the GCD.

```
main_program{ // Compute GCD of m,n, where m > n >0.
  int m,n;
  cout << "Enter the larger number (must be > 0): ";  cin >> m;
  cout << "Enter the smaller number (must be > 0): "; cin >> n;

  while(m % n != 0){
    int Remainder = m % n;
    m = n;
    n = Remainder;
  }
  cout << "The GCD is: " << n << endl;
}
```

7.8 CORRECTNESS OF LOOPING PROGRAMS

It should be intuitively clear that the programs discussed in this chapter are correct. However, intuition can be deceptive, and as we have discussed earlier, it is better to cross-check. In this section we discuss how to argue the correctness of programs more formally.

In arguing the correctness of `repeat` loop-based programs we can typically state what progress we expect will happen in each iteration, and this can be expressed in the plan that we write and prove (Section 4.2.2). The argument for proving the correctness of programs that use `while/for` loops is more complex than the argument for `repeat` based programs (Section 4.2.2). This is because we do not know in general how many times a `while/for` loop will execute. Thus the argument must also show that the loop eventually terminates.

The proof argument for `while/for` loops tends to typically have a two parts: a *loop invariant*, and a *potential*. We will explain these notions next, and along with the explanation we will prove the correctness of the GCD program given above.

7.8.1 Loop Invariant

A loop invariant is an assertion about the values taken by variables in a program that must be true before and after every iteration of the loop. The term invariant is to be understood like the conservation

principles of physics, e.g. the total energy of the system is the same after the experiment as it was before. A loop invariant is similar in spirit to the *plan* we discussed in Section 4.2.2.

We next describe the invariants needed to prove the correctness of our GCD program. Suppose m_0, n_0 are the values given as input for variables `m, n`. We will prove the following invariants.

> **Invariant 1:** (Before and after each iteration of the loop) The GCD of `m, n` remains unchanged, i.e. equals the GCD of m_0, n_0.
>
> **Invariant 2** (Before and after each iteration of the loop) We have `m > n > 0`.

Invariant 2 will also be useful to show that the program terminates and has no errors along the way. Invariant 1 will show that the correct answer is produced.

Invariants are proved using mathematical induction, as you might expect. We prove the second invariant first. When control reaches the loop, for the first iteration, the variables `m, n` will have values m_0, n_0. We will have $m_0 > n_0 > 0$ assuming the user followed our instructions. Thus, the base case for the induction is established. So now suppose that at the beginning of some tth iteration, `m > n > 0`. We will prove that at the end of the tth iteration and hence at the beginning of the $t + 1$th iteration (if any), we will continue to have `m > n > 0`. So let us consider the execution of the loop. The loop test computes `m % n`. This operation is valid only if `n > 0`. But we assumed that `n > 0` at the beginning of the iteration. Hence, the remainder `m % n` will be well defined and computed properly without the possiblity of division by 0. If `m % n` is 0, then the loop body will not be entered; there will not be any $t + 1$th iteration, and so there is nothing to prove. So assume that the remainder is positive. In this case the loop body is entered. The first statement in the body sets `Remainder` to the remainder. Note now that `Remainder` must have a smaller value than the divisor, `n`. The last two statements of the loop respectively assign the values of `n` and `Remainder` to `m` and `n`. Thus at the end of the loop `m` will have a larger value than `n`, as required.

The first invariant, i.e. the GCD of the new values being the same as the GCD of the old values, is a direct consequence of Euclid's theorem. However, we will state the proof more formally. As before, the proof uses mathematical induction. When control reaches the loop for the first iteration, the variables `m, n` have values m_0, n_0. Thus, the GCD of `m, n` is obviously the same as the GCD of m_0, n_0. So the base case holds. So consider what happens after t iterations. We execute the loop test. The loop test requires us to divide `m` by `n`. If the loop test fails, then there is no $t + 1$th iteration and hence nothing to prove. However if the loop test succeeds, then we enter the loop. In the loop, we assign values to `m, n` exactly as per Euclid's theorem. Hence, the GCD of `m, n` is unchanged after the assignment, though the values of `m, n` have themselves changed.

7.8.2 Potential

Intuitively, it should be clear that the values of `m, n` will keep reducing and hence eventually the loop test must succeed. We now observe it formally. The value of `n` in the next iteration is the current value of `m%n`, which is clearly smaller than the current value of `n`. Hence in each iteration, the value of `n` decreases by at least 1. But `n` is guaranteed to be always positive, i.e. never drop to 0 or become negative. Hence, the number of iterations cannot be more than the value n_0 typed in by the user at the beginning of the program. Thus, the loop must terminate sometime!

The key idea in this argument is the observation that some quantity must decrease by at least some fixed amount, but the nature of the loop body is such that the quantity cannot decrease below a certain

threshold. This establishes that the number of iterations must be finite, otherwise the quantity will have decreased below the threshold. In the case of GCD, it is convenient to choose as potential the value of `n`. But in other programs, there will be other choices, sometimes creativity will be needed to define a suitable potential.

This quantity is metaphorically called the *Potential*, inspired by the notion of potential energy in physics.

7.8.3 Correctness

Given appropriate invariants and a suitable potential, the correctness proof is almost done. Usually it is only a matter of tying up some loose ends.

When the GCD program terminates, we know from the invariant that GCD of the current values of `m, n` must be the same as the GCD of m_0, n_0. Since the loop test must have failed just before termination, we know that `Remainder == 0`, i.e. `m % n == 0`. But then the GCD must be `n`, which is indeed what we print. Thus, we have established correctness.

We also note that our program runs correctly even if the user disregards our instructions and types in the smaller number first and larger second. The invariants and the analysis remains correct!

7.8.4 Additional Observations Regarding the GCD Program

We note that the above argument can be sharpened to get a stronger bound on the number of iterations needed by the GCD program. Let m_i, n_i denote the value of `m` and `n` respectively at the beginning of the ith iteration. Let R_i denote the value of `Remainder` calculated in the ith iteration. Then we know:

1. At the end of the ith iteration, the variable `n` gets the value `m % n`, i.e. $n_{i+1} = R_i = m_i \bmod n_i$. Further, `m` gets the value of the variable `n`. Thus, $m_{i+1} = n_i$. Note that the remainder modulo n_i must be smaller than n_i. Thus, we know that $n_{i+1} < m_{i+1}$.
2. In iteration $i + 1$, the computation is similar. Thus, we get $n_{i+2} = R_{i+1} = m_{i+1} \bmod n_{i+1}$. Suppose q is the quotient when m_{i+1} is divided by n_{i+1}. Thus, $m_{i+1} = qn_{i+1} + R_{i+1}$. But $m_{i+1} > n_{i+1}$, and so $q \geq 1$. Thus, $m_{i+1} \geq n_{i+1} + R_{i+1}$.

Thus, we have $n_i = m_{i+1} \geq n_{i+1} + R_{i+1} = n_{i+1} + n_{i+2}$. But we know that `n` decreases in each iteration, and so $n_{i+1} > n_{i+2}$. Thus, $n_{i+1} + n_{i+2} > 2n_{i+2}$. Thus we have proved that $n_i > 2n_{i+2}$. Thus, we have established that the value of `n` drops by a factor at least 2 in 2 iterations. But `n` never drops below 1. Thus, the number of iterations is at most $2 \log_2 n_0$, where n_0 is the value of `n` as typed in by the user.

7.8.5 Correctness of Other Programs

Other programs, e.g. primality, could also be proved correct in a similar manner.

7.9 CONCLUDING REMARKS

Looping is a very important operation in programming. It is tricky because we need to match the pattern of repetition in our problem with the pattern of repetition provided in the looping statement. Looping using the statements `while`, and `for` is tricky also because of the need to ensure that the loop eventually terminates, something that was not needed for loops based on `repeat`.

In this chapter, we have seen how various problems can be solved using the `while` loop as well as the `for` loop. There were were many variations depending upon whether we used `break`, or replicated code. Later on in the book, we will see even more ways of expressing some of the programs we have seen in this chapter. As we have indicated, each way of writing loops has some advantages and disadvantages. One may be more readable or less readable, another may avoid duplication of code, and yet another may be less efficient because it does unnecessary work. Another consideration is *naturalness*: is a certain way of writing code more consistent with how you might think about the problem? So the choice of how to express a program is in the end a subjective choice. So you should develop your own taste in this regard.

The `while` and `for` loops are trickier than `repeat` loops. This is because it is possible to make a programming error and write `while/for` loops that do not terminate. Hence, we must be more careful in using these loops as compared to `repeat` loops. This complexity is reflected in the manner in which we argue the correctness. You may observe that the correctness argument for `repeat` did not need to have anything like a potential because the `repeat` loops are guaranteed to terminate no matter what.

We have remarked earlier that proving programs can be tedious for large programs. However, we will emphasize that even if you don't do full proofs, you should write down invariants and potentials for each non-trivial program that you write.

EXERCISES

1. Write a program that prints a conversion table from Centigrade to Fahrenheit, say between $0°$C to $100°$C. Write using `while` and also using `for`.
2. Suppose we are given n points in the plane: $(x_1, y_1), \ldots, (x_n, y_n)$. Suppose the points are the vertices of a polygon, and are given in the counterclockwise direction around the polygon. Write a program using a `while` loop to calculate the perimeter of the polygon. Also do this using a `for` loop.
3. Write a program that returns the approximate square root of a non-negative integer. For this exercise define the approximate square root to be the largest integer smaller than or equal to the exact square root. Your are expected to not use the built-in `sqrt` or `pow` commands, of course. Your program is expected to do something simple, e.g. check integers in order $1, 2, 3, \ldots$ to see if it qualifies to be an approximate square root.
4. Suppose some code contains some `while` statements. Show how you can replace the `while` statements by `for` statements without changing the output produced by the code.
5. Add a "Stop" button to the turtle controller of Section 6.4.1. Modify the program so that it runs until the user clicks on the stop button. Also there should be no limit on the number of commands executed by the user.
6. Write a program that prints out the digits of a number starting with the least significant digit, going on to the most significant. Note that the least significant digit of a number `n` is simply `n % 10`.
7. Write a program that takes a number `n` and prints out a number `m` which has the same digits as `m`, but in reverse order.
8. A natural number is said to be a palindrome if the sequence of its digits is the same whether read left to right or right to left. Write a program to determine if a given number is a palindrome.

9. Write a program that takes as input a natural number x and returns the smallest palindrome larger than x.

10. Add checks to the GCD code to ensure that the numbers typed in by the user are positive. For each input value you should prompt the user until she gives a positive value.

11. Write a program that takes a natural number and prints out its prime factors.

12. Write a program that reads in a sequence of characters, one at a time, and stops as soon as it has read the contiguous sequence of characters 'a', 'b', 'r', 'a', 'c', 'a', 'd', 'a', 'b', 'r', 'a', i.e. the string "abracadabra". *Hint*: After you have read a certain number of characters, what exactly do you need to remember? Do you need to remember the entire preceding sequence of characters, even the last few characters explicitly? Figure out what is needed, and just remember that in your program. This is a difficult problem.

13. Let $x_1, \ldots, x_n$ be a sequence of integers (possibly negative). For each possible subsequence $x_i, \ldots, x_j$ consider its sum S_{ij}. Write a program that reads in the sequence in order, with n given at the beginning, and prints out the maximum sum S_{ij} over all possible subsequences.
Hint: This is a difficult problem. However, it will yield to the general strategy: figure out what set of values $V(k)$ we need to remember having seen the first k numbers. When you read the $k+1$th number, you must compute $V(k+1)$ using the number read and $V(k)$ which you computed earlier.

CHAPTER 8

Computing Common Mathematical Functions

In this chapter, we will see ways to compute some common mathematical functions, such as trigonometric functions, square roots, exponentials and logarithms. We will discuss the Newton–Raphson method for finding roots, and use it for computing square roots. Incidentally, this method of finding square roots (but not the generalization due to Newton and Raphson) was known to the Babylonians, as long ago as 1500 BC!

The main statement in all the programs of the chapter will be a looping statement. You could consider this chapter to be an extension of the previous, giving more ways in which loop statements can be used.

Some of the material in this chapter requires somewhat deep mathematics. We will state the relevant theorems, and try to explain intuitively why they might be true. The precise proofs are outside the scope of this book.

8.1 TAYLOR SERIES

Suppose we wish to compute $f(x)$ for some function f, such as say $f(x) = \sin(x)$. Suppose we know how to compute $f(x_0)$ for some fixed x_0. Suppose that the derivative f' of f and the derivative f'' of f' and so on exist at x_0, and we can evaluate these. Then if x is reasonably close to x_0 then $f(x)$ equals the sum of the *Taylor series* of f at x_0. The ith term of the Taylor series is $f^{i\prime}(x_0)(x - x_0)^i/i!$, in which $f^{i\prime}$ is the function obtained from f by taking derivative i times. Thus, we have

$$f(x) = f(x_0) + f'(x_0)(x - x_0) + f''(x_0)\frac{(x - x_0)^2}{2!} + f'''(x_0)\frac{(x - x_0)^3}{3!} + \cdots \tag{8.1}$$

In the typical scenario, we only compute and sum the first few terms of the series, and that gives us a good enough estimate of $f(x)$. The general theory of this is discussed in standard mathematics texts and is outside our scope. However, you may recognize the first two terms as coming from a tangent approximation of the curve, as shown in Figure 8.1. The value of $f(x)$ equals (the length of) FD. We approximate this by FC, which in turn is FB + BC = EA + (BC/AB)AB = $f(x_0) + f'(x_0) \cdot (x - x_0)$. In other words,

$$f(x) \approx f(x_0) + f'(x_0) \cdot (x - x_0) \tag{8.2}$$

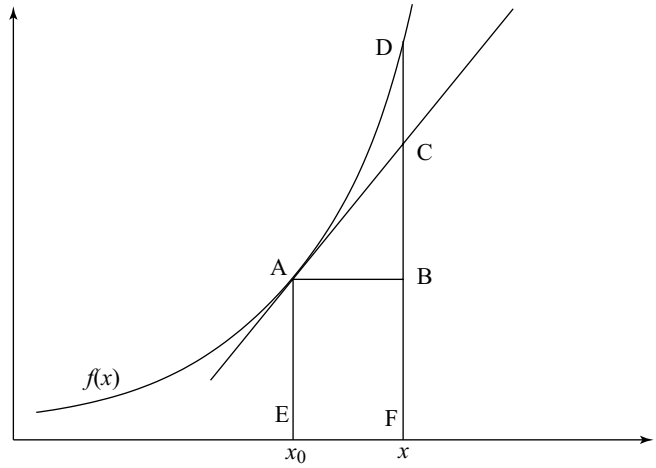


Fig. 8.1 *Tangent approximation of f at A, $(x_0, f(x_0))$*

The Taylor series is a very sophisticated version of this approximation.[1] Note that the Taylor series is often written as

$$f(x_0 + h) = f(x_0) + f'(x_0)h + f''(x_0)\frac{h^2}{2!} + f'''(x_0)\frac{h^3}{3!} + \cdots \tag{8.3}$$

which is obtained by writing $x = x_0 + h$. The tangent approximation, Equation (8.2) becomes

$$f(x_0 + h) \approx f(x_0) + h \cdot f'(x_0) \tag{8.4}$$

If we choose $x_0 = 0$, we get the McLaurin series, which is

$$f(x) = f(0) + f'(0)x + f''(0)\frac{x^2}{2!} + f'''(0)\frac{x^3}{3!} + \cdots \tag{8.5}$$

This form is very commonly used, and perhaps easier to remember.

8.1.1 Sine of an Angle

As an example, consider $f(x) = \sin(x)$, where x is in radians. We use the Mclaurin form. We know that $f'(x) = \cos(x)$, $f''(x) = -\sin(x)$, $f'''(x) = -\cos(x)$ and so on. Noting $\cos(0) = 1, \sin(0) = 0$, we know the exact value of every derivative, it is either 0, 1 or -1. Thus, we get

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!} - \cdots$$

Here, the angle x is in radians. When a series has alternating positive and negative terms, and the terms get closer and closer to 0 for any fixed x, then it turns out that the error in taking just the first k terms is at most the absolute value of the $k+1$th term. The kth term of our series is $(-1)^{k+1}x^{2k-1}/(2k-1)!$. Thus, if we want the error to be ϵ then we should ensure $x^{2k+1}/(2k+1)! \leq \epsilon$.

[1] You might be familiar with the formula $s(t) = ut + \frac{1}{2}at^2$, in which $s(t)$ is the distance covered in time t by a particle moving at a constant acceleration a, with initial velocity u. Note that $u = s'(0)$, $a = s''(0)$ and with this substitution the formula can be written as $s(t) = s(0) + s'(0)t + s''(0)\frac{t^2}{2}$, which resembles the Taylor series in the first three terms.

We have already seen how to sum series (Section 4). Clearly, we will need a loop, in the kth iteration of which we will calculate the series to k terms. We must terminate the loop if the last added term is smaller than our target ϵ. We can calculate the kth term t_k from scratch in the kth iteration, but it is useful to note the following relationship:

$$t_k \;=\; (-1)^{k+1}\frac{x^{2k-1}}{(2k-1)!} \;=\; t_{k-1}\left((-1)\frac{x^2}{(2k-2)(2k-1)}\right)$$

provided $k > 1$. If $k = 1$ then $t_k = 1$, of course, and we don't use the above relationship. Thus, within the loop we only compute the terms for $k = 2, 3, \ldots$ as needed. Thus, our code becomes

```
main_program{
  double x;  cin >> x;

  double epsilon = 1.0E-20, sum = x, term = x;

  for(int k=2; abs(term) > epsilon; k++){
              // Plan: term = t_{k-1}, sum = sum of k-1 terms
    term *= -x * x /((2*k-2)*(2*k-1));
    sum += term;
  }

  cout << sum << endl;
}
```

The command `abs` stands for absolute value, and returns the absolute value of its argument.

8.1.2 Natural Log

Consider $f(x) = \ln x$, the natural logarithm of x. One way of defining it is

$$\ln x = \int_1^x \frac{1}{u}du$$

So from this, we can find its Taylor series. Clearly $f'(x) = 1/x$. $f''(x) = -1/x^2$ and so on. It is convenient to use $x_0 = 1$. Thus, we get

$$\ln 1 + h = h - \frac{h^2}{2} + \frac{h^3}{3} - \frac{h^4}{4}\cdots$$

A very important point to note for this series is that the series is valid only for $-1 < h \leq 1$. Even so, note that you can indeed use the series to calculate $\ln x$ for arbitrary values of x. Simply observe that $\ln x = 1 + \ln \frac{x}{e}$. Thus, by factoring out powers of e we will need to use the series only on a number smaller than 1.

8.1.3 Remarks

In general, the terms of the Taylor/McLaurin series increase with x. Thus, it is best to keep x small if possible. For example, suppose we wish to compute $\sin(100.0)$. One possibility is to use the previous program specifying 100.0 as input. A better way is to subtract as many multiples of 2π as possible, since we know that $\sin(x + 2n\pi) = \sin(x)$ for any integer n. In fact identities such as $\sin(x) = -\sin$

$(\pi - x)$ can be used to further reduce the value used in the main loop of the program. In fact, noting that the Taylor series for $\cos(x)$ is

$$\cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \cdots$$

we can in fact compute the sine using $\sin(x) = \cos(\pi/2 - x)$ if $\pi/2 - x$ happens to be smaller in absolute value than x.

8.2 NUMERICAL INTEGRATION

We consider another way of computing $\ln x$ given x. Recall the definition:

$$\ln x = \int_1^x \frac{1}{u} du$$

In other words, $\ln x$ is the area under the curve $y = 1/u$ between $u = 1$ and $u = x$. So we can compute $\ln x$ if we can compute the area!

Well, we are not going to compute the area exactly, but we will approximate it. In general suppose we wish to compute the area under a curve $f(u)$ from some $u = p$ to some $u = q$. We proceed as follows.

1. Divide the area into n vertical strips of equal width $w = \frac{q-p}{n}$, where n is a parameter to be chosen. Thus the ith strip where $i = 0, \ldots, n-1$ lies between $u = p + iw$ and $u = p + (i+1)w$.
2. Estimate the area of the ith strip to be $w \cdot f(p + iw)$. Notice that this is a reasonable estimate if f does not vary much within the strip. This will happen if w is small, which in turn will happen if n is large.
3. Add up the estimates and return that as the area. Thus, the area returned is

$$\sum_{i=0}^{n-1} wf(p + iw) = w \sum_{i=0}^{n-1} f(p + iw)$$

It should be intuitively clear that if we increase n, the strip width w will reduce, and along with it the error in the estimate of the area. You may recall from calculus courses that the integral is in fact defined as the limit of the above sum as n approaches ∞.

We next give the program to evaluate the above sum for $f(u) = \frac{1}{u}$, between $p = 1$ and $q = x$, which will give us an approximate value for $\ln x$. As you can see, i in the above sum will naturally serve as a control variable for our `for` loop. We will take each successive term of the series and add it into a variable `area` which we first set to 0. The following is the complete program.

```
main_program{
  double x; cin >> x;        // will calculate ln(x)
  int n; cin >> n;           // number of rectangles to use
  double w = (x-1)/n;        // width of each rectangle
  double area = 0;           // will contain ln(x) at the end.
  for(int i=0; i < n; i++)
     area = area + w /(1+i*w);
  cout << "Natural log, from integral: "<< area << endl;
  cout << "Natural log, built-in command: "<< log(x) << endl;
}
```

In the last line, we have used the built-in C++ command `log` which can be invoked as `log(x)` and which then returns the value of the natural logarithm. This command uses some code probably more sophisticated than what we have written above, and it guarantees that the answer it returns will be correct to as many bits as your representation. Thus, in the last line we can check how well our approximation works. You are encouraged to run the program for different values of n and see how close we can get to the exact answer (as computed using `log`).

8.2.1 An Improvement: The Midpoint Rule

Turns out that there is a better way to approximate the area of each strip. Specifically, instead of the "left height" $f(p + iw)$ of the ith strip, we use the height in the middle, i.e. $f(p + iw + \frac{w}{2})$. Thus, we get the following estimate for the area.

$$w \sum_{i=0}^{n-1} f(p + iw + \frac{w}{2})$$

This estimate is only marginally different from the previous one, but it indeed gives much better approximation. The proof of this is outside the scope of this book, but you can of course experimentally check whether what we say is true.

You may also think of other ways to get better estimates of the areas of the strips. For example, one way is to consider each strip to be a trapezium, bounded by the corners $(p + iw, 0)$, $(p + iw, f(p + iw))$, $(p + (i + 1)w, 0)$ $(p + (i + 1)w, f(p + (i + 1)w))$. More sophisticated rules have also been invented, e.g. Simpson's rule. The exercises invite you to experiment with these rules.

8.2.2 Increasing the Number of Strips

You may think that the sophistication of the mathematical approximation does not matter, because "we can always increase n, the number of strips". This expectation is naive. The problem is that on a computer numbers are represented only to a fixed precision. For example, if we use the `float` representation then every number is correct only to a precision of about 7 digits. If you add n numbers each containing an (additive) error of ϵ, then the error in the sum could become $n\epsilon$, assuming all errors were in the same direction. Even assuming that the errors are random, it is possible to show that the error will be proportional to $\sqrt{n}\epsilon$. In other words, if you add $n = 10000$ numbers, each with an error of about $\epsilon = 10^{-7}$, your total error is likely to have risen to about 10^{-5} (if not to 10^{-3}). Thus, we should choose n large, but not too large. The exercises ask you to experiment to find a good choice. Of course, if you use `double`, then you should be able to go to larger values of n. Even higher if you use `long double`.

The alternative to increasing n is to use better mathematical approximations, i.e. the midpoint rule instead of the basic rule, or even better approximations such as Simpson's rule.

8.3 BISECTION METHOD FOR FINDING ROOTS

A *root*, or a *zero* of a function f is a value x_0 such that $f(x_0) = 0$. In other words, a point where the plot of the function touches the x-axis. Many problems can be expressed as finding the roots of an equation. For example, suppose we want to find the square root of 2. Then instead we could ask for the roots of the polynomial $f(x) = x^2 - 2$. Clearly, if $f(x) = 0$ then we have $x^2 - 2 = 0$, i.e. $x = \pm\sqrt{2}$ and this would give us the square root of 2. So finding roots is a very important mathematical problem.

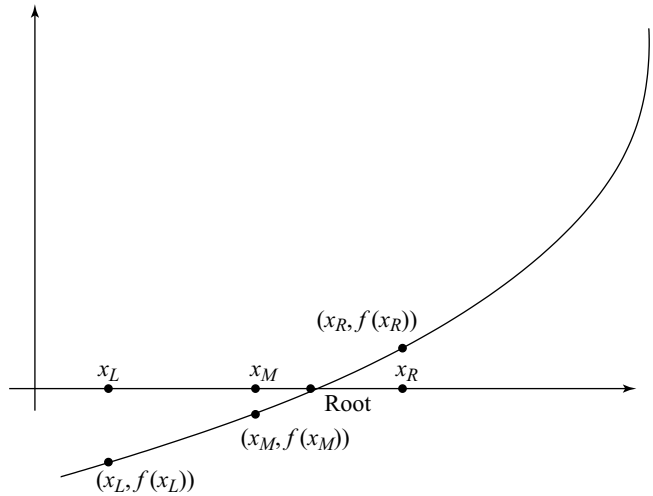


Fig. 8.2 *Bisection method. Next we will have* $x_L = x_M$.

In this section, we will see a very simple method for finding roots $approximately$. The method will require that (a) we are given values $x_L \leq x_R$ such that $f(x_L)$ and $f(x_R)$ have opposite signs, and (b) f is continuous between x_L and x_R. These are fairly minimal conditions, for example for $f(x^2) = x^2 - 2$ we can choose $x_L = 0$ giving $f(x_L) = -2$, and $x_R = 2$, giving $f(x_R) = 2$. Clearly x_L, x_R satisfy the conditions listed above.

Because f is continuous, and has opposite signs at x_L, x_R, it must pass through zero somewhere in the (closed) interval $[x_L, x_R]$. We can think of $x_R - x_L$ as the degree of uncertainty (or maximum error), in our knowledge of the root. Getting a better approximation merely means getting a smaller interval, i.e. getting x_L, x_R such that $x_R - x_L$ is smaller. If the size of the interval is very small, we can return either endpoint as an approximate root. So the main question is: can we somehow pick better x_L, x_R given their current values?

A simple idea works. Consider the interval midpoint: $x_M = (x_L + x_R)/2$. We compute x_M and find the sign of $f(x_M)$. Suppose the sign of $f(x_M)$ is different from the sign of $f(x_L)$. Then we set $x_R = x_M$. Clearly the new values x_L, x_R satisfy our original requirements. If the sign of x_M is the same as the sign of x_L, then it must be different from the sign of x_R. In that case (see Figure 8.2) we set $x_L = x_M$. Again the new values of x_L, x_R satisfy our requirements. Hence in each case, we have reduced the size of the interval, and thus reduced our uncertainty. Indeed if we want to reduce our error to less than some ϵ, then we must repeat this process until $x_R - x_L$ becomes smaller than ϵ. Then we would know that they are both at a distance at most ϵ from the root, since the root is inside the interval $[x_L, x_R]$.

The code is then immediate. We write it below for finding the square root of 2, i.e. for $f(x) = x^2 - 2$.

```
main_program{              // find root of f(x) = x*x - 2.
  double xL=0,xR=2; // invariant: f(xL),f(xR) have different signs.
  double xM,epsilon;
  cin >> epsilon;
  bool xL_is_positive, xM_is_positive;
```

```
  xL_is_positive = (xL*xL - 2) > 0;
  // Invariant: xL_is_positive gives the sign of f(x_L).

  while(xR-xL >= epsilon){
    xM = (xL+xR)/2;
    xM_is_positive = (xM*xM -2) > 0;
    if(xL_is_positive == xM_is_positive)
      xL = xM;  // does not upset any invariant!
    else
      xR = xM;  // does not upset any invariant!
  }
  cout << xL << endl;
}
```

8.4 NEWTON–RAPHSON METHOD

We can get a faster method for finding a root of a function f if we have a way of evaluating $f(x)$ as well as its derivative $f'(x)$ for any x. To start off this method, we also need an initial guess for the root, which we will call x_0. Often, it is not hard to find an initial guess; indeed in the example we will take, almost any x works as the initial guess.

In general, the Newton–Raphson method takes as input a current guess for the root, say x_i. It returns as output a (hopefully) better guess, say x_{i+1}. We then compute $f(x_{i+1})$, if it is close enough to 0, then we report x_{i+1} as the root. Otherwise, we repeat the method with x_{i+1} to get, hopefully, an even better guess x_{i+2}.

The process of computing x_i given x_{i+1} is very intuitive. We use the tangent approximation to f as given in equation (8.2). Thus, we get $f(x) \approx f(x_i) + f'(x_i) \cdot (x - x_i)$, assuming $x - x_i$ is small. In this equation we could choose x to be any point, including the root. So let us choose x to be the root. Then $f(x) = 0$. Thus, we have $0 \approx f(x_i) + f'(x_i) \cdot (x - x_i)$. Or in other words, $x \approx x_i - \frac{f(x_i)}{f'(x_i)}$. Notice that the right-hand side of this equation can be evaluated. Thus, we can get an approximation to the root! This approximation is what we take as our next candidate.

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} \tag{8.6}$$

That is all there is! Figure 8.3 shows what happens graphically. The point A with coordinates $(x_i, 0)$ represents our current estimate of the root. We draw a vertical line from A up to the function taking us to the point B, having coordinates $(x_i, f(x_i))$. At B, we draw a tangent to the function f, and the tangent intersects the x axis in point C. If we consider the tangent to be a good approximation to f, then the root must be point C. Indeed, we take the x coordinate of C to be our next estimate x_{i+1}. Thus, we have

$$x_{i+1} = x_i - AC = x_i - \frac{AB}{AB/AC} = x_i - \frac{f(x_i)}{f'(x_i)}$$

which is what we obtained earlier arguing algebraically. At least in the figure, you can see that our new estimate is better, indeed the point C has moved closer to the root as compared to the point A. It is possible to argue formally that if x_i is reasonably close to root to start with, then x_{i+1} will be even closer. Indeed, in many cases, it can be shown that the number of bits of x_i that are correct essentially

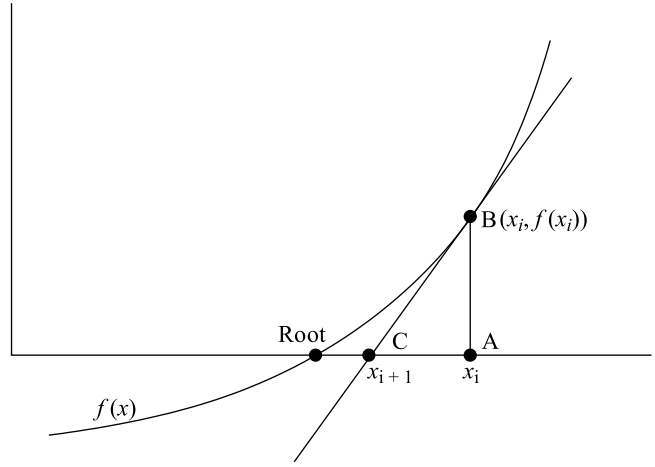


Fig. 8.3 *One step of Newton–Raphson*

double in going to x_{i+1}. Thus, a very good approximation to the root is reached very quickly. The proof of all this is not too hard, at least for special cases, but beyond the scope of this book.

We now show how the Newton–Raphson method can be used to find the square root of any number y. As with the bisection method, we must express the problem as that of finding the root of an equation: $f(x) = x^2 - y$. We also need the derivative, and this is $f'(x) = 2x$. The update rule, $x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$ in this case becomes

$$x_{i+1} = x_i - \frac{x_i^2 - y}{2x_i} = \frac{1}{2}(x_i + \frac{y}{x_i})$$

Next we need an initial guess. The standard idea is to make an approximate plot of the function, and choose a point which appears close to the root. In this case, it turns out that almost any initial guess is fine, except for 0, because at 0 the term y/x_i would be undefined. So for simplicity, we choose $x_0 = 1$. So we are ready to write the program. The basic idea is to maintain a variable `xi` representing the current guess. We will update `xi` in each iteration using the above rule, and initialize `xi` to 1.

```
main_program{
  double xi=1, y;  cin >> y;
  repeat(10){
    xi = (xi + y/xi)/2;
  }
  cout << xi << endl;
}
```

This program will run a fixed 10 iterations, and calculate the estimates $x_1, x_2, \ldots, x_{10}$, starting with $x_0 = 1$. But we can also run a number of iterations depending upon how much error we wish to tolerate.

This is slightly tricky. If the actual root is x^*, then the error in the current estimate x_i is $|x_i - x^*|$. Indeed, if we exactly knew the error, i.e. the value $v = |x_i - x^*|$, we could directly compute the root by noting that $x^* = x_i \pm v$. So we need to make an estimate for the error. A common estimate is $f(x_i)$. Indeed, $f(x_i)$ is the vertical distance of the point $(x_i, 0)$ to the curve f whereas the exact error, $x_i - x^*$ is the horizontal distance of the point $(x_i, 0)$ to the curve. Indeed, when the vertical distance becomes

0, so would the horizontal. So our program can terminate when the error estimate $f(x_i) = x_i^2 - y$ becomes small, i.e. when `xi*xi-y` becomes smaller than some threshold.

```
main_program{
  double y; cin >> y;
  double xi=1;
  while(abs(xi*xi - y) >0.001){
    xi = (xi + y/xi)/2;
  }
  cout << xi << endl;
}
```

In the above code, we have used the built-in function `abs` which returns the absolute value of its argument.

8.5 CONCLUDING REMARKS

This chapter has introduced several new ideas, though no new programming langauge features.

We saw some general techniques for computing mathematical functions. We saw that if the function and its derivatives are easy to evaluate for some values of the argument, then the Taylor series of the function can often be used to give an algorithm that evaluates the function. Another idea was: if a function is defined by an integral, then we can evaluate it by numerically evaluating the integral.

We also discussed methods to find the roots of an equation. This is important intrinsically, as you will see in the Exercises. It is also important because the problem of computing mathematical functions can be formulated as a problem of finding roots. We discussed the *bisection method*. This is a very general method, useful even if we only know how to evaluate the function f whose roots we seek. If we can evaluate the first derivative of f in addition, we saw that we can use the Newton–Raphson method. The Newton-Raphson method is one of the most powerful tools for finding roots and evaluating mathematical functions. The bisection method and the Newton-Raphson method work by starting with a guess, and improving it in successive iterations. Such methods are called *Iterative methods*.

We will come back to the ideas presented here in chapters 19 and 29.

EXERCISES

1. Write a program to find $\ln x$ for arbitrary x using the Taylor series. Check your answer by using the built-in `log` command.
2. Write down the Taylor series for $f(x) = e^x$, noting that $f^{i\prime}(x) = e^x$. It is convenient to expand around $x_0 = 0$, i.e. consider the McLaurin series. This series is valid for all values of x, however, it is a good idea to use it on as small values of x as possible. Write a program to compute e^x, and check against the built-in command `exp`.
3. Run the program for computing $\ln x$ from Section 8.2 for various choices of n and see how the result varies. For what value of n do you get an answer closest to the `log` function of C++?
4. Another way to estimate the area under a curve is to use trapeziums rather than rectangles. Thus the area under a curve $f(u)$ in the interval $[p, q]$ will be approximated by the area of the trapezium

with corners $(p, 0), (p, f(p)), (q, f(q)), (q, 0)$. This area is simply $(f(p) + f(q))(q - p)/2$. Apply this to each strip while computing the natural logarithm.

5. Simpson's rule gives the following approximation of the area under the curve of a function f:

$$\int_a^b f(x)dx \quad \approx \quad \frac{b-a}{6}\left(f(a) + 4f\left(\frac{a+b}{2}\right) + f(b)\right)$$

Use this rule for each strip to get another way to find the natural logarithm.

6. Suppose we are given n points in the plane: $(x_1, y_1), \ldots, (x_n, y_n)$. Suppose the points are the vertices of a polygon, and are given in the counterclockwise direction around the polygon. Write a program to calculate the area of the polygon. *Hint 1*: Break the area into small triangles with known coordinates. Then compute the lengths of the sides of the triangles, and then use Heron's formula to find the area of the triangles. Then add up. *Hint 2*: Break the boundary of the polygon into two parts, an up facing boundary and a down facing boundary. Express the area of the polygon in terms of the area under these boundaries. Then perform numerical integration.

7. Children often play a guessing game as follows. One child, Kashinath, picks a number between 1 and 1000 which he does not disclose to another child, Ashalata. Ashalata asks questions of the form "Is you number between x and y?" where she can choose x, y as she wants. Ashalata's goal is to ask as few questions as possible and determine the number that Kashinath picked. Show that Ashalata can guess the number correctly using at most 10 questions. *Hint*: Use ideas from the bisection method.

8. Write a program to find $\arcsin(x)$ given x.

9. Consider a circuit in which a voltage source of $V_{CC} = 1.5$ volts is applied to a diode and a resistance of $R = 1$ ohm connected in series, Figure 8.4. The current I through a diode across which there is a potential drop of V is

$$I = I_S(e^{V/(nV_T)} - 1)$$

where I_S is the reverse saturation current of the diode, V_T is the thermal voltage which is about 25 mV at room temperature (300 Kelvin), and n is the ideality factor. Suppose the diode we are using has $n = 1$ and $I_S = 30$ nA. Write a program that finds the current. Use your program to also find the current when the voltage source is reversed.

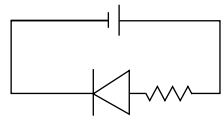


Fig. 8.4 *Diode circuit*

10. Consider the problem of finding the roots of $f(x) = x^3 - x/2 + 1/4$. See what happens using the Newton-Raphson method for guesses for the initial value. In particular, try $x_0 = 1$ and $x_0 = 0.5$. Can you solve this using the bisection method?

CHAPTER 9

Functions

In the preceding chapters, we have seen programs to do many things, from drawing polygons and miscellaneous pictures to calculating income tax and finding the greatest common divisor (GCD). It is conceivable that we will want to write more and more complex programs in which some such calculations, e.g. finding the GCD, are needed at many places. One possibility is to copy the code for a calculation in as many places as is required. This doesnt seem too elegant, and is also error-prone. Wouldnt it be nice, if for each frequently used operation you could somehow construct a "command" that could then be used wherever you want in your program? Just as we have a command `sqrt` for computing the square root, or commands for computing the trigonometric ratios (Section 1.5) can we build a `gcd` command that will compute the GCD of two numbers when demanded? This can be done, and how to build such commands is the subject of this chapter.

The term *function* is used in C++ to denote what we have so far informally called a command. In some languages the terms *procedure* or *subprogram* are also used. In what follows, we will use the term *function*.

We will also discuss *references* and *pointers*. These notions will be important in connection with functions, and in other contexts.

9.1 DEFINING A FUNCTION

Suppose that we indeed need to frequently compute the GCD, and so would like to have a function which does this. It is natural to choose the name `gcd` for this function. It could take two numbers as arguments, and return their GCD, which could then be used. As an example, suppose you wanted to find the GCD of 24 and 36, and also the GCD of 99 and 47. If we had a `gcd` function as described, then we could write a very simple main program as follows.

```
main_program{
  int a=36,b=24,c=99,d=47;
  cout << gcd(a,b) << endl;
  cout << gcd(c,d) << endl;
}
```

```
int gcd             // return-type function-name
  (int m, int n)    // parameter list: (parameter-type
                       parameter-name ...)
{                   // beginning of function body
  while(m % n != 0){
    int Remainder = m % n;
    m = n;
    n = Remainder;
  }
  return n;
}                 // end of function body
```

Fig. 9.1 *Definition of a function to compute the GCD*

The text `gcd(a,b)` and the text `gcd(c,d)` are said to be *calls* or *invocations* of the function `gcd` with `a, b` the *arguments* to the first call, and `c,d` the arguments to the second call.

Next we discuss how to define the function `gcd`. Basically, in the definition, we must specify what needs to happen when the function is called during execution, e.g. for the call `gcd(a,b)` in the main program above. In essence, the idea is to have a small program run, sort of in the background, for computing the GCD. This program, which we will refer to as a *subprogram* must be given the inputs, (in the present case, the values of the numbers whose GCD is to be computed), and some mechanism must be established for getting back the result of the computation (in the present case, the computed GCD) to the main program. While the sub-program runs, the main program must simply wait. The main program can resume after the sub-program finishes its work.

Figure 9.1 shows the code for defining `gcd`. The simplest way to use the definition is to place it in the same file as the main program given earlier, before the main program. If you compile and run that file, then it will indeed print 12 and 1, the GCD respectively of 24, 36 and 99, 47, as you expect. The requirement that the function definition be placed before the main program is similar to the requirement that a variable must be defined before it is used. We can relax this requirement slightly, as will be seen in Section 11.2.7.

In general, a function definition has the form:

```
type-of-return-value function-name (parameter1-type parameter1-name,
  parameter2-type parameter2-name, ...){
  body
}
```

The definition begins with `type-of-return-value`, which indicates the type of the value returned by the function. In the GCD example, the function computes and evaluates the GCD, which has type `int`, so our definition (Figure 9.1) mentions this.

Next is `function-name`, the name of the function being defined. In our example, we chose to call our function `gcd`, so that is what the code states. Any valid identifier (Section 3.1.1) can be chosen as a function name.

Next is a parenthesised list of the parameters to the function, together with their types. In our case, there are two parameters, `m,n` both of type `int`.

Finally, comes the code, `body`, that is used to compute the return value. The `body` is expected to be a sequence of statements, just as you would expect in any main program. It can contain declarations of

Activation frame of `main_program`	Activation frame of `gcd(a,b)`
`a` : 36	`m` : 36
`b` : 24	`n` : 24
`c` : 99	
`d` : 47	

(a) After copying arguments

Activation frame of `main_program`	Activation frame of `gcd(a,b)`
`a` : 36	`m` : 24
`b` : 24	`n` : 12
`c` : 99	`Remainder` : 12
`d` : 47	

(b) At the end of the first iteration of the loop in `gcd`

Fig. 9.2 *Some snapshots from the execution of* `gcd(a,b)`

variables, conditional statements, looping statements, everything that can be present in a main program. However, there are two additional features. The code in the body can refer to the parameters, as if they are variables. Further, the the body must contain a `return` statement, which we explain shortly. We note that the `body` of the definition in Figure 9.1 is taken substantially from the program developed in Section 7.7.

9.1.1 Execution: Call by Value

Consider our `gcd` function and main program. While executing the main program, suppose that control arrives at the call `gcd(a,b)`. We describe the general rule that determines what happens, and also mention what happens in our specific case.

1. The arguments to the call are evaluated. In our case, it simply means fetching the values of the variables `a,b`, viz. 36,24. But in general, the arguments could be arbitrary expressions which would have to be evaluated.
2. The execution of the calling subprogram, i.e. the subprogram which contains the call, `main_program`, in this case, is suspended. The calling subprogram will be resumed later. When resumed, the execution will continue from where it was suspended.
3. Preparations are made to start running a subprogram. The subprogram will execute the code given in the `body` of the function. The subprogram must be given a separate area of memory so that it can have its own variables. It is customary to refer to this area as the *activation frame* of the function call. Immediately, space is allocated in the activation frame for storing the variables corresponding to the parameters of the function.
   Thus, in our case, an activation frame is created corresponding to the call `gcd(a,b)`. The `gcd` function has two parameters, `m, n`. So variables, `m` and `n` will be created in the activation frame.
4. The value of the first argument is copied to the memory associated with the first parameter. The value of the second argument to the second parameter, and so on.

Thus, in our case, 36 will be copied into the variable `m`, and 24 into the variable `n` in the activation frame created for the call `gcd(a,b)`. Figure 9.2(a) shows the state of the memory at this time. We have referred to the memory area used by `main_program` as its activation frame. This is customary.

5. Now the `body` of the called function is executed. The `body` must refer to variables or parameters stored only in the activation frame of the call.[1] If space needs to be reserved for variables etc., it is done only inside the activation frame of the call.
   Thus, in case of our program, the code may refer to the parameters `m, n`. The code *cannot* refer to variables `a,b,c,d` because they are not in the activation frame of `gcd(a,b)`. When the first statement of the `body` is executed, it causes the creation of the variable `Remainder`. The space for this is allocated in the activation frame of the call. Such variables are said to be *local* to the call.
6. The `body` of the function is executed until a `return` statement is encountered. The expression following `return` is called the `return-expression` and its value is sent back to the calling program. The value sent back is considered to be the value of the call in the calling program.
   In our case, the execution of the function happens as follows. In the first iteration of the loop, `m, n` have values 36,24. At the end of this iteration, the values become 24,12. The state of the memory at this point is shown in Figure 9.2(b). In the next iteration, `Remainder` becomes 0, and so the `break` statement is executed. Thus the control exits from the loop, and `return` is reached. The `return-expression` is `n` which has value 12. This value is sent back to the calling program.
7. The activation frame created for the call is not needed any longer, and is destroyed, i.e. that area is marked available for general use.
8. The calling program resumes execution from where it had suspended. The returned value is used as the value of the call itself.
   In our case, the call was `gcd(a,b)`, and its value is required to be printed. Thus the value returned, 12, will be printed. After this, the next `cout` statement will be executed in which we will encounter the second call to `gcd`. This will cause an activation frame to be created again etc.

In this model of executing function calls, only the values of the arguments are sent from the calling program to the called function. For this reason, this model is often termed as *call by value*. We will see another model later on.

It is worth considering what happens on the second call to `gcd`, i.e. the call `gcd(c,d)` in the code. The same set of actions would repeat. A new activation frame would be created for this call, and very likely it would use the same memory as was used for the activation frame of the previous call. The point to be noted is that each call requires some additional memory, but only for the duration of the execution of the call.

9.1.2 Names of Parameters and Local Variables

We have already said that when a function call executes, it can only access the variables (including the parameters) in its activation frame. In particular, the variables in the calling program (in this case `main_program`) cannot be accessed. So it is perfectly fine if variables in the calling program and

[1] We will modify this a bit later.

the called function have the same name! Note further that when the calling program is executing, the activation frame of the called function does not even exist, so there is no question of any confusion.

9.2 NESTED FUNCTION CALLS: LCM

Suppose now that you wish to develop a program to compute the Least Common Multiple (LCM) of two numbers. This is easily done using the following relationship between the LCM, L, and the GCD, G of two numbers m, n:

$$L = \frac{m \times n}{G}$$

It would, of course, be nice to write a function for the LCM, so that we could invoke it whenever needed, rather than having to copy the code. We could use the above relationship, but that would require us to compute the GCD itself. Does it mean that we need to rewrite the code for computing the GCD inside the function to compute LCM? Not at all. We can simply call the `gcd` function, since we have already written it! So here is how we can define a function to compute the LCM.

```
int lcm(int m, int n){
  return m*n/gcd(m,n);
}
```

The execution of `lcm` follows the same idea as in our discussion earlier for `gcd`. Suppose `lcm` is called in by a main program as follows.

```
main_program{
  cout << lcm(36, 24) << endl;
}
```

When we execute the main program, we will need to run a subprogram for `lcm`, which involves creating the activation frame for this call. As this subprogram executes, we will encounter the expression `gcd(m,n)` with `m, n` taking the values 36, 24. To process this call, we will need to start a subprogram for `gcd`. So at this point, we will have 3 activation frames in memory, one for `main_program`, one for `lcm(36,24)` and another for `gcd(36,24)`. This is perfectly fine! When the subprogram for `gcd(36,24)` finishes, then the result, 12, will be sent back to the subprogram for `lcm(36,24)`. The result 12, will be used as the value of the call `gcd(m,n)`. Thus, the expression `m*n/gcd(m,n)` can now be evaluated to be `36*24/12=72`. This will in fact be the value that the subprogram `lcm(36,24)` returns back to `main_program`. At this point, the computation of `main_program` will resume with the received value.

9.3 THE CONTRACT VIEW OF FUNCTION EXECUTION

While it is important to know how a function call executes, while designing functions, a different, metaphorical view is useful.

The idea is to think of a function call as giving out a *contract* to get a job done. We think of the main program as an agent doing its work as described in its program. Suddenly, the agent encounters a statement such as `lcm(36,24)`. Rather than doing the work required to compute `lcm(36,24)` itself, the main program agent engages another agent. This agent is the subprogram for the call `lcm(36,24)`. The main program agent sends the input data to the subprogram agent, and waits for the

result to be sent back. This is not unlike engaging a tailor, giving the tailor the cloth and measurements, and waiting for the tailor to send back a shirt.

The similarity extends further. There is nothing to prevent the tailor from further (sub) contracting out the work to others. It so happens, that stitching the collar of a shirt is a specialized job, which most tailors would in fact contract out to collar-specialists. Thus, it is possible that we may be waiting for the tailor to send us back the shirt, and the tailor might be waiting for the collar specialist to send back a collar. Notice that this is very similar to `main_program` waiting for `lcm(36,24)` which in turn is waiting for `gcd(36,24)`.

9.3.1 Function Specification

A key point to be noted from the tailor example above is that when we ask for a shirt to be stitched, we generally do not worry about how the tailor will do it. The tailor may do all the work, or subcontract it out further to one or more craftsmen—that is not our concern. We merely focus on the promise that the tailor has made to us—that a shirt will be delivered to us. We don't worry about how the tailor does it, but we merely hold the tailor to deliver us a good shirt (and at the right time and price, as per what has been agreed). If we tried to worry about what our tailor should be doing, and what our accountant should be doing, and what our doctor should be doing, we would probably go mad!

Likewise, when we call a function in our program, we do not think of how exactly it will get executed. We merely ask: what exactly is being promised in the execution of this function? The promise, is actually both ways, like a contract and is customarily called the *specification* of the function. The specification of `gcd` could be as follows:

> A call `gcd(m,n)` returns the greatest common divisor of `m,n`, where `m,n` must be positive integers.

You will notice that the specification lays down the responsibilities of both the calling program, and the called program.

1. **Responsibilities of the calling program:** To supply only positive integers as arguments. Notice that C++ already prevents you from supplying fractional values when you declare the type of the parameters to be `int`. However, nothing prevents a calling program from supplying negative values or 0. The specification says that the programmer who wrote the function `gcd` makes no guarantees if you supply 0 or negative values. The conditions that the input values are required to satisfy are often called the *pre-conditions* of the function. In addition, the calling program might also have to deal with *post-conditions*, as will be discussed in Section 9.4.
2. **Responsibilities of the called program:** If the calling program fullfills its responsibilities (i.e. the arguments satisfy the preconditions), and only if the calling program fullfills its responsibilities, is the called program obliged to do whatever was promised. There is no telling what will happen if the preconditions are not satisfied. Thus, in case of `gcd`, if a negative value or zero is supplied: nonsense values may be returned, or the program may never terminate, or terminate with an error.

It is extremely important to clearly write down the specification of a function. You may sometimes avoid doing so, thinking that the specification is obvious. But it may not be so! For example, a more general definition of GCD might allow one of the numbers to be zero, in which case the other number is defined to be the GCD. If this is the definition a user is familiar with, he/she might supply 0 as the value of the second parameter `n`. This will certainly cause the program to terminate because of a division by

zero in the very first step of our code. To prevent such misunderstandings, it is best to write down the specifications in full detail.

The natural place to write down the specification is immediately before the function body. So your function for `gcd` should really look like the following.

```
int gcd(int L, int S)
// Function for computing the greatest common divisor of integers L,S.
// PRE-CONDITION: L, S > 0
{
...
}
```

Please get into the habit of writing specifications for all the functions that you write. Note that in the specification it is important to not write *how* the function does what it does, but only *what* the function does, and for what preconditions.

A description of how the function does what it does, often referred to as the description of the *implementation* of the function is also important. But this should be kept separate from the specification. The description of *how* can be in a separate document, or could be written as comments in the body of the code of the function. For example, the following comment might be useful to explain how the `gcd` function works.

```
// Note the theorem: If n divides m, then GCD(m,n) = n.
//           If n does not divide m, then GCD(m,n) = GCD(n, m mod n)
```

This comment could be placed at the beginning of the loop.

9.4 FUNCTIONS THAT DO NOT RETURN VALUES

Every function/command need not return a value. You have already seen such functions, e.g. `forward`, which does not return any value, but only causes the turtle to move forward. The function `forward` is said to cause a *side-effect*, i.e. it causes the graphics canvas to change "on-the-side", instead of returning a value up front.

You can also write functions which cause side effects rather than return values. For example, you might wish to write a function which draws a polygon with a given number of sides, and having a certain given side length. Clearly, it must take two arguments, an integer giving the number sides, and a `double` giving the side length. Suppose we name it `drawPolygon`. The function does not return any value, so we are required to specify the return type in the definition to be `void`. Also, since nothing is being returned, we merely write `return` with no value following it.

```
void drawPolygon(int nsides, double sidelength)
// draws polygon with specified sides and specified sidelength.
// PRE-CONDITION: The pen must be down, and the turtle must be
// positioned at a vertex of the polygon, pointing in the clockwise
// direction along an edge.
// POST-CONDITION: At the end the turtle is in the same position and
// orientation as at the start.  The pen is down.
{
```

```
  for(int i=0; i<nsides; i++){
    forward(sidelength);
    right(360.0/nsides);
  }
  return;
}
```

Note the precondition: it states where the polygon is drawn in comparison to where the turtle is pointing. Similarly, we should mention where the turtle is at the end, this will be needed in order to know how to draw subsequently. A condition such as this one, which will be true after the execution of the function, is said to be a *post-condition* of the function. A post-condition is also a part of the specification.

9.4.1 Side Effects and Values

It is possible to have a function which not only returns a value, but has side effects too. Further, side effects need not only be to the graphics canvas, you could also print messages and read from the keyboard as a side effect. Here is a function that does all these things: it reads a value in a given range from the keyboard and returns it.

```
int readFromRange(int small, int large)
// Returns number between small and large, both inclusive.
// PRE-CONDITION: small must be no larger than large.
{
  int num;
  do{
    cout << "Give a number between " << small << " and "
    << large << ": ";
    cin >> num;
  } while( small > num || num > large );

  return num;
}
```

Our main program could then contain a call to this such as

```
int val = readFromRange(1,10);
```

This would ask the user for a number between 1 and 10, and repeat if the user gave a number outside the range. Eventually, if the user does give a number in the range that number would be returned and placed in `val`.

9.5 A TEXT-DRAWING PROGRAM

We would like to develop a program using which it is possible to write on the screen using our turtle. For example, we might want to write “IIT MUMBAI”. How should we organize such a program?

A natural (but not necessarily the best, see the exercises) way of organizing this program is to have a separate function for writing each letter. For example, we will have a function `drawI` for drawing

the letter 'I'. Suppose we decide that we will write in a simple manner, so that the letter 'I' is just a line, without the serifs (horizontal lines at the top and bottom), i.e. as **I**.

What is the specification of `drawI`? Clearly, it must draw the line as needed. But where should the line get drawn? This must be mentioned in the specifications. It is tempting to say that the line will get drawn at the current position of the turtle, in the direction the turtle is pointing. Is this really what we want? Keep in mind that usually you will not want to draw just one letter, but a sequence of letters. So it is important to bring the turtle to a convenient position for drawing subsequent letters. And what is that convenient position?

Suppose we think of each letter as being contained inside a rectangle. It is customary to call this rectangle the *bounding-box* of the letter. Then we will make it a convention that the turtle must be brought to the bottom left corner of the bounding box, and point towards the direction in which the writing is to be done. Where would we like the turtle to be at the end of writing one character so that the next character can be written easily? Clearly, the most convenient final position is pointing away from the right bottom corner, pointing in the direction of writing. We must also clearly state in the precondition whether we expect the pen to be up or down. Is the inter-character space to be a part of the bounding box? If so, a further question arises: is it on both sides of the character or only on one side? We should not only answer these questions, but must also include the answers in the specification.

Based on the above considerations, `drawI` could be defined as follows.

```
void drawI(double ht, double sp){
/*Draws the letter I of height ht, leaving sp/2 units of space on
  both sides. Bounding box includes space.
  PRECONDITION: The turtle must be at the left bottom of the
  bounding-box in which the character is to be drawn, facing
  in the direction of writing.  Pen may be up or down.
  POSTCONDITION: The turtle is at the bottom right corner of the
  bounding-box, facing the writing direction, with pen up.  */

  forward(sp/2);
  penDown();
  left(90);
  forward(ht);
  penUp();
  left(180);
  forward(ht);
  left(90);
  forward(sp/2);
  return;
}
```

Functions for other letters are left as exercise for you. So assume that you have written them. Then to write our message, our main program could be as follows.

```
main_program{
  int ht=100, sp=10;
  turtleSim();
  left(90);     // turtle is pointing East at the beginning.
```

```
  drawI(ht,sp);
  drawI(ht,sp);
  drawT(ht,sp);
  forward(sp);
  drawM(ht,sp);
  drawU(ht,sp);
  drawM(ht,sp);
  drawB(ht,sp);
  drawA(ht,sp);
  drawI(ht,sp);
}
```

Note that there are local variables named `ht` and `sp` in the main program, as well as the functions have parameters called `ht` and `sp`. As we have mentioned earlier, this is acceptable. When the function is being executed, the execution refers only to its activation frame, and hence the variables in the main program are not visible. When the main program is executing, the activation frame of the functions is not even present, so there is no confusion possible.

9.6 SOME DIFFICULTIES

There are a few seemingly simple things we cannot do using our current notion of a function. For example, we might want to write a function which takes as arguments the Cartesian coordinates of a point and returns the Polar coordinates. This is not immediately possible because a function can only return one value, not two. Another example is: suppose we want to write a function called `swap` which exchanges the values of two integer variables. Suppose we define something like the following.

```
void swap(int a, int b){  // will it work?
  int temp;
  temp = a;
  a = b;
  b = temp;
}
```

If we call this by writing `swap(p,q)` from the main program, we will see it does not change the values of `p,q` in the main program. The reason for this is that when `swap` executes, it does exchange the values `a,b`, but `a,b` are in the activation frame of `swap`, and their exchange does not have any effect on the values of `p,q` which are in the activation frame of the main program.

As a third example, consider the mark-averaging program from Chapter 7. An important step in this program is to read the marks from the keyboard and check if the marks equal 200. If the marks equal 200, then the loop needs to terminate. Here is an attractive way to write the program.

```
main_program{
  double nextmark, sum=0;
  int count=0;

  while(read_marks_into(nextmark)){  // will this work?
    sum = sum + nextmark;
```

```
    count = count + 1;
  }

  cout << "The average is: " << sum/count << endl;
}
```

Our hope is that we can write a function `read_marks_into` that will behave in the following manner. It will read the next mark into the variable given as the argument, and also return a true or false depending upon whether the reading was successful, i.e. true if the value read was not 200, and false if it was. But what we have learned so far does not allow us to write this function: The value of the argument `nextmark` will be copied to the parameter of the function, but will not be copied back.

It turns out that all the three problems listed above have a nice solution in C++. This solution is based on another way of passing arguments to function, called *call by reference*. We will see this next.

Following that we will see how the problem is solved in the C language. As you might know, C++ is considered to be an enhanced version of C. Nevertheless, there are a number of reasons for discussing the C solution. First of all, it is good to know the C solution because C is still in use, substantially. Also, you may see our so-called C solution in C++ programs written by someone, because essentially all C programs are also C++ programs. Second, the C solution uses the notion of *pointers*, which are needed in C++ also. Finally, the C solution is in fact a less magical version of the call by reference solution of C++. So in case you care, the C solution might help you understand "what goes on behind the scenes" in call by reference.

9.7 CALL BY REFERENCE

The idea of call by reference is simple: when you make a change to a function parameter during execution, you want the change to be reflected in the corresponding argument? Just say so and it is done! The way to "say so" is to declare the parameter whose value you want to be reflected as a *reference parameter*, by adding an `&` in front of the name of the parameter. So here is how we might write the function to convert from Cartesian to polar.

```
void Cartesian_To_Polar(double x, double y, double &r, double &theta){
  r = sqrt(x*x + y*y);
  theta = atan2(y,x);
}
```

In this function, `r` and `theta` have been declared to be reference parameters. No storage is allocated for a reference parameter in the activation frame of the function, nor is the value of the corresponding argument copied. Instead, during the execution of the function, a reference parameter directly *refers* to the corresponding argument. Hence, whatever changes the function seems to make to a reference parameter are really made to the corresponding argument directly.

This can be called in the normal way, possibly as follows.

```
main_program{
  double x1=1.0, y1=1.0, r1, theta1;
  Cartesian_To_Polar(x1,y1,r1,theta1);
  cout << r1 << ' ' << theta1 << endl;
}
```

Here is how the call `CartesianToPolar(x1,y1,r1,theta1)` executes. The values of `x1,y1` are copied to the corresponding parameters `x,y`. However, as mentioned, the values of `r1, theta1` are not copied. Instead, all references to `r, theta` in the function are deemed to be references to the variables `r1,theta1` instead! Thus, as `CartesianToPolar` executes, the assignments in the statements `r=...` and `theta=...` get made to `r1` and `theta1` directly. So indeed, when the function returns, the variable `r1` would contain $\sqrt{1+1} = \sqrt{2} \approx 1.4142$, and `theta1` would contain $\tan^{-1} 1 = \pi/4 \approx 0.785$, and these would be printed out.

The function to swap variable values can also be written in a similar manner.

```
void swap2(int &a, int &b){
  int temp;
  temp = a;
  a = b;
  b = temp;
}
```

This can be called as follows.

```
main_program{
  int x=5, y=6;
  swap2(x,y);
  cout << x << " " << y << endl;
}
```

Both the arguments of `swap2` are references, and so nothing is copied into the activation area of `swap2`. The parameters `a,b` refer directly to `x,y`, i.e. effectively we execute

```
temp = x;
x = y;
b = temp;
```

This will clearly exchange the values of `x,y`, so at the end "6 5" will be printed.

Our last example is the `read_marks_into` function discussed in Section 9.6.

```
bool read_marks_into(int &var){
  cin >> var;
  return var != 200;
}
```

This definition will work as desired with the main program given in Section 9.6. When the function executes, the first line will read a value into `var`. But `var` is a reference to the corresponding argument `nextmark`, and hence the value will in fact be read into `nextmark`. The expression `var != 200` is true if `var` is not 200, and false if it is 200. So the while loop in the main program will indeed terminate as soon as 200 is read. Continuing the discussion at the end of Section 7.2, we note that perhaps this is the nicest way of writing the mark averaging program: we do not duplicate any code, and yet the loop termination is by checking a condition at the top, rather than using a `break` statement in the body.

9.7.1 Remarks

Call by reference is very convenient. However, two points should be noted.

The manner by which we specify arguments of a function in a call is the same, no matter whether we use call by value or by reference for a parameter. This makes it hard to read and understand code. When we see a function call, we need to either find the function definition or declaration (Section 11.2.1) to know which of the arguments, if any, correspond to reference parameters, and hence might change when the function returns. The C language solution which uses pointers, discussed next, does not have this drawback. On the other had it has other drawbacks, as you will see.

If a certain parameter in a function is a reference parameter, then the corresponding argument must be a variable. For example, we cannot write `swap2(1,z)`. This would make `a,b` refer to `1,z` respectively and then the statement `a = b;` in the function would be equivalent to `1 = z;`, which is meaningless. So supplying anything other than a variable is an error if the corresponding parameter is a reference parameter. However, also see the discussion about `const` reference parameters in Section 17.2.1.

9.7.2 Reference Variables

In the discussion above, we noted that a reference parameter should be thought of as just a name, what it is a name of is fixed only when we make the call. In a similar manner, we can have reference variables also.

```
int x;
int &r = x;
x = 10;
cout << r << endl;
r = 20;
cout << x << endl;
```

The first statement defines the variable `x`. The second statement declares a reference `r`, hence the `&` before the name. In the declaration itself we are obliged to say what the name `r` is a reference to. This is specified after =. Thus, the second statement declares the integer reference `r` which is a reference to `x`, or just another name for the variable `x`. In the fourth statement, we print `r`. Since `r` is another name for the variable `x`, the value of that, 10, gets printed. In the fifth statement, we assign 20 to `r`, but since `r` is just a name for `x`, it really changes the value of `x`. Finally, this changed value, 20, gets printed in the last statement.

The utility of reference variables will become clear later, in Section 28.5.

9.8 POINTERS

We first discuss pointers in general, and then say how they are helpful in solving the problems of Section 9.6.

We know from Section 2.6.1 that memory is organized as a sequence of bytes, and the `i`th byte is supposed to have address `i`, or be at address `i`. When memory is allocated to a variable, it gets a set of consecutive bytes. The address of the first byte allocated to the variable is considered to be the address of the variable.

9.8.1 "Address of" Operator &

C++ provides the unary operator & (read it as "address of") which can be used to find the address of a variable. It is called unary because it is to be applied only to a single expression, as you will see. Yes, this is the same character that we used to mark a parameter as a reference parameter, and there is also a binary operator & (Appendix C). But you will be able to tell all these apart based on the context. Here is a possible use of the unary &.

```
int p;
cout << &p;  // In this operator & is applied to p.
```

This will print out the address of p. Note that the convention in C++ is to print out addresses in hexadecimal, so you will see something that begins witn 0x, which indicates that following it is a hexadecimal number. Note that in hexadecimal each digit takes value between 0 and 15. Thus, some way is needed to denote values 10, 11, 12, 13, 14, 15, and for these the letters a,b,c,d,e,f respectively are used.

9.8.2 Pointer Variables

We can store addresses into variables if we wish. But for this we need to define variables of an appropriate type. For example, we may write

```
int p=15;
int *r;                // not ``int multiplied by r''!  See below.
r = &p;
cout << &p << " " << r << endl;
```

The first statement declares a variable p as usual, of type int. The next statement could be read as (int*) r;, i.e. int* is the type and r is the name of the declared variable. The type int* is used for variables which are expected to contain addresses of int variables. This is what the third statement does, it stores the address of the int type variable p into r. If you execute this code, you will see that the last statement will indeed print identical hexadecimal numbers.

Figure 9.3 schematically shows a snapshot of memory showing the effect of storing the address of p into r. In this, we have assumed that p is allocated bytes 104 through 107, and r is allocated bytes 108 through 111. The address of p, 104, appears in bytes 108 through 111, as a result of the execution of r = &p;.

Address	Content	Remarks
104 105 106 107	15	Allocated to p
108 109 110 111	104	Allocated to r

Fig. 9.3 *Picture after executing* r = &p;

Likewise we may write

```
double q;
double* s = &q;
```

Here we have declared and initialized `s` in the same statement. Note that `double*` and `int*` are different types, and you may not write `s = &p;` or `r = &q;`.

Variables `r, s` are said to be `pointers` to `p, q`. In general, variables of type `T*` where `T` is a type are said to be pointer variables.

Finally, even though we use integers to number memory locations, it is never necessary in C++ programs to explicitly store a specific constant, say 104, into a pointer variable. If you somehow come to know that 104 is the address of a certain variable `v`, and so you want 104 stored in some pointer variable `w`, then you can do so by writing `w = &v;`, without using the number 104 itself. In fact, it is a compiler error in C++ to write something such as `w=104`, where `w` is a pointer, e.g. of type `int*`. Because you don't need to write this, if you actually do, it is more likely to be a typing mistake. So the compiler flags it as an error.

Finally, it should be noted that C++ declarations are a bit confusing. The following

```
int* p, q;       // both pointers?
```

declares `p` to be a pointer to `int`, while `q` is simply an `int`. Even if you put no space between `int` and `*` in the above statement, the `*` somehow "associates" with `p` than with `int`. With this understanding, perhaps we should read a declaration `int *p;` to mean: "the type of the content of `p` is `int`" which is another way of saying that `p` is a pointer to `int`.

If you want both `p, q` to be pointers, you must write

```
int *p, *q;     // both pointers!
```

9.8.3 Dereferencing Operator *

If we know the address of a variable, we can get back that variable by using the *dereferencing operator*, `*`. Very simply put, the unary `*` can be considered to be the inverse of `&`. The character `*` also denotes the multiplication operator, and is also used in declaration of pointer variables, but it will be clear from the context which operator is meant.

Formally, suppose `xyz` is of type `T*` and has value `v`. Then we consider the memory at address `v` to be the starting address of a variable of type `T`, and `*xyz` denotes this variable. The unary `*` is to be read as "content of", e.g. an expression such as `*xyz` above is to be read as "content of `xyz`".

Thus, consider the following code.

```
int p=15, *r;
r = &p;
*r = 22;
int m;
m = *r;
```

The second statement placess the address of `p` into `r`. Now `*r` denotes a variable of type `int` stored at address `&p`. But `p` is exactly such a variable. Hence, `*r` denotes the variable `p` itself. Thus, if `*r` were to appear on the left-hand side of an assignment statement, we would really be storing a value into `p`. If `*r` appeared on the right hand side of an assignment, or in an expression, we would be using the

value of p in place of the expression *r. Thus in the third statement, we would store 22 into p. In the last statement, we would store the value of p, 22 in this case, into m.

9.8.4 Use in Functions

We first note that functions can take data of any type as arguments, including types such as int* or double*. Thus, we can write a function to compute the polar coordinates given Cartesian as follows.

```
void CartesianToPolar(double x, double y, double* pr, double* ptheta){
  *pr = sqrt(x*x + y*y);
  *ptheta = atan2(y,x);
}
```

This could be called as follows.

```
main_program{
  double r,theta;
  CartesianToPolar(1.0, 1.0, &r, &theta);
  cout << r << " " << theta << endl;
}
```

Let us first make sure that the types of the arguments in the call and the parameters in the function definition match. The first and second parameters, x, y are required to be a double, and indeed the first and second arguments are both 1.0, of type double. The third parameter pr is of type double*. The third argument is the expression &r, which means the address of r. Since r is of type double, the type of &r is indeed double*, and hence the type of the third argument and the third parameter match. Similarly the type of the fourth argument &theta is also seen to match the type double* of the fourth parameter. So clearly our program should compile without errors.

Let us see how this will execute. When the function CartesianToPolar is called, none of the parameters are reference parameters, and so all arguments have to be copied first. So 1.0 is copied to the parameter x in the activation frame of CartesianToPolar. The second argument 1.0 is copied to y. The third argument &r is copied to pr, and finally the fourth argument &theta is copied to ptheta.

Then the body of the function is executed. The first statement is *pr = sqrt(x*x + y*y);. The right hand side evaluates to $\sqrt{2}$, because x and y are both 1. This value is to be placed in the variable denoted by the left hand side. Now *pr is interpreted exactly as described in Section 9.8.3. Given that pr is of type double*, the expression *pr denotes that double variable whose address appears in pr. But we placed the address of r of the main program in pr. Hence, *pr denotes the variable r of the main program. Hence the statement *pr = sqrt(x*x + y*y);, even if it appears in the code of CartesianToPolar will store $\sqrt{2}$ into the variable r of main.

Next let us consider the statement *ptheta = atan2(y,x);. Since y,x are both 1, the arctangent will be found to be $\pi/4 \approx 0.785$. Reasoning as before, the expression *ptheta will denote the variable theta of the main program. Thus 0.785 will be stored in theta of main. After this the call will terminate. When the execution of main resumes, $\sqrt{2}$ and 0.785 would get printed by the last statement in main.

We next consider the swap function. It should be clear to you now what we should do: instead of using the variables as arguments, we should use their addresses. Here is the function.

```
void swap(int* pa, int* pb){
  int temp;
  temp = *pa;
  *pa = *pb;
  *pb = temp;
}
```

It may be called by a main program as follows.

```
main_program{
  int x=5, y=6;
  swap(&x,&y);
  cout << x << " "<< y << endl;
}
```

The arguments to the call are `&x, &y`, having type `int*`, because `x,y` are of type `int`. Thus they match the types of the parameters of the function. Thus, our program will be compiled correctly.

So let us consider the execution. The address of `x` will be copied into `pa`, and the address of `y` into `pb`. Thus we may note that `*pa` in `swap` will really refer to the variable `x` of the main program, and `*pb` in `swap` will refer to the variable `y` of the main program. The statement `temp = *pa;` will cause the value of `x` to be copied to `temp`. Next, the statement `*pa = *pb` will cause the value of `y` to be copied to `x`. Finally, the statement `*pb = temp;` will cause the value in `temp`, i.e. the value that was in `x` at the beginning to be copied to `y` (which is what `*pb` denotes). After this the function call completes. The main program then resumes and will print the exchanged values, 6 and 5.

The changes required to the main program and the function `read_marks_into` for mark averaging are left as exercises.

9.8.5 Reference vs Pointers

You have seen that there are two ways of writing the functions `Cartesian_To_Polar`, `swap` and `read_marks_into`. Which one is better?

Clearly, the functions are easier to write with call by reference. So that is clearly to be recommended in C++ programs.[2]

9.9 RETURNING REFERENCES AND POINTERS

It is possible to return references and pointers from functions. But this has to be done with care. We will explain the idea, but for interesting use of it you will have to wait till Section 18.5 and Section 21.3.7.

First of all, in order to return a reference to a type `T` the return type of the function must be given as `T&`, as you might expect. Here is an example.

[2] You are probably wondering: when a function executes, and some parameter is a reference parameter, how does the computer know what variable the parameter refers to? A simple answer is: at the time of the call, C++ automatically sends the address of the variables referred to by the reference parameters to the function activation frame. Also, during the function execution, C++ itself dereferences the address of the reference variables, and gets to the variables as needed. So in other words, the operations of sending addresses and dereferencing them that had to be manually written out in C are performed "behind the scenes" by C++.

```
int &f(int &x, int &y){
  if(x > y) return x;
  else return y;
}
main_program{
  int p=5, q=4;
  f(p,q) = 2;                    // function call appears on the left!
  cout << p <<' '<< q << endl;
  cout << f(p,q) << endl;
}
```

This main program contains a function call on the left-hand side of an expression! Normally, a function returns a value, and there is no notion of assigning one value to another value! However, we can certainly assign a value to a reference, and hence a function that returns a reference can indeed be on the left-hand side of an assignment statement. Thus, the statement will execute by evaluating the value of the right-hand side, which will then be placed in the variable that the left-hand side refers to.

So consider the execution of `f(p,q)=2;` in the above code. Noting that `p,q` are being passed by reference, the references `x,y` will refer respectively to the variables `p,q` of the main program. Thus, the expression `x>y` is identical to `p>q`, where `p,q` are variables in the main program. Thus, `x>y` will evaluate to true. Thus, the statement `return x;` will be executed. Because the function has return type reference to int (`int&`), the value of `x` (which is the value of the variable it refers to) is not returned, but the reference itself is returned. Since `x` is a reference to `p`, the call returns a reference to `p`. But then the main program statement `f(p,q) = 2;` is equivalent to `p = 2;`. Hence, the statement will cause `p` to become 2. Thus the print statement will print "2 4".

Note that a call to a function that returns a reference does not *have* to be on the left hand side of the assignment. Indeed, in the last line, we print `f(p,q)`. When this call executes, `x,y` refer to `p,q` as you might expect. The comparison `x>y` is now false, because `p` now has value 2. Thus, `y` is returned by reference, i.e. a reference to `q` is returned. Thus, the last statement, `cout « f(p,q) « endl;` is equivalent to the statement `cout « q « endl;`, and hence the value 4 will get printed.

9.9.1 Dangling References and Dangling Pointers

It should be noted that returning a reference can be dangerous.

```
double &h(){
  double x = 5;
  return x;
}
main_program{
  h() = 7;
}
```

The function call `h()` will return a reference to the local variable `x` of the function. Unfortunately, after the function returns, the local variable will no longer exist! Such a reference is often called a *dangling* reference. Thus, in the main program it will be incorrect to either modify `h()` or get its value. Note that most C++ compilers will not give any errors for the code above. However, this code is definitely incorrect.

Similar ideas apply to returning pointers. The analogue of the first example above is as follows.

```
int *f(int *x, int *y){
  if(*x > *y) return x;
  else return y;
}
main_program{
  int p=5, q=4;
  *f(&p,&q) = 2;
  cout << p <<' '<< q << endl;
  cout << *f(&p,&q) << endl;
}
```

In this, the call `f(&p,&q)` returns the address of a variable, so it can be dereferenced and then we can either modify the variable or get its value. Thus, the first print statement will print "2 4" and the second will print 4 as before.

Analogously, it is incorrect to return the address of a variable that will be deallocated by the time the address can be used.

```
double *h(){
  double x = 5;
  return &x;
}
main_program{
  *h() = 7;
}
```

The function call `h()` returns the address of the variable `x` local to the function call. The variable is destroyed when the function call returns. Hence, it is incorrect to use `h()` in any way in `main`.

9.10 CONCLUDING REMARKS

When you write complex programs, you may often find that the same sequence of operations is needed at different places in the program. Instead of writing the same code at different places, C++ allows you to put that code into a *function*, which can then be called wherever it is needed. This is the main reason for defining functions. There will be other reasons, as we will see later. Functions can return values, and also cause side effects.

A function should be viewed as a packaged *software component*. When desigining a function, you should very clearly write down its specification, i.e. what it does. In the specification, it is acceptable to specify preconditions, i.e. say something like "Function `f` will return ... only if parameter `x` satisfies ...". If the function has a side effect, then it is important to specify the post conditions also. The motivation for writing down all this is to divide responsibility:

- The specification provides clear guidelines to the developer as to what to develop.
- The user of the function can use it without worrying about how exactly the function does its work.

Even if the function is written and used by the same programmer, by writing specifications, the programmer's mental stress is reduced. While writing the function the programmer only concentrates

on the specification and does not worry about the different situations in which the function will be used, because that is implicitly captured in the specification. While writing the rest of the program, the programmer does not worry about what goes on inside the function—the specification spells out what the end result of the computation will be. Note finally that it is important that the name chosen for the function reflect its purpose.

The calling program can specify the data which the function can use in its execution in different ways. The simplest mechanism for this *parameter passing* is *call by value*, i.e. the values to be used in the execution are copied into variables in the activation frame of the called function. Another mechanism is *call by reference*. In this, the calling function specifies the variables which the called function can use. The key point here is that the called function can make modifications to the specified variables. The calling function will see the changes after the called function returns. Note that if a function modifies the values of the reference parameters, this is a side effect, and this must be very well documented.

We also saw the notion of pointers. It is possible to designate parameters of a function as being pointers. When a call is made to such a function, the called functions specifies an address, which is then copied into the corresponding parameter in the called function. The called function can dereference the pointer if it wishes and get access to the data stored in the variable to which the pointer points. The called function can then even modify the value of the variable if it so chooses. To this extent, passing the pointer to a variable is similar to passing by reference. You may wonder what happens if a pointer is passed by reference, see Exercise 8.

EXERCISES

1. Write a function that prints the GCD of two numbers.
2. Modify the function `drawPolygon` so that it returns the perimeter of the polygon drawn, in addition to drawing the polygon.
3. Write a function to find the cube root of a number using Newton's method. Accept the number of iterations as an argument.
4. Write a function to determine if a number is prime. It should return `true` or `false`, i.e. be of type `bool`.
5. Write the function `read_marks_into` and the main program for mark averaging using pointers.
6. A key rule in assignment statements is that the type of the value being assigned must match the type of the variable to which the assignment is made. Consider the following code:

```
int *x,*y, z=3;

y = &x;
z = y;
y = *x;
```

Each of the assignments is incorrect. Can you guess why? If not, write the code in a program, compile it, and the compiler will tell you!

7. What does the following code print?

```
void f(int x, int &y, int *z){
  cout << x++ <<' '<< y++ <<' '<< ++*z << endl;
}
main_program{
  int a=10, b=20, c=30;
  f(a,b,&c);
  f(c,a,&b);
}
```

8. You may wonder whether it is possible to pass a pointer by reference. It is indeed possible, as shown in the following code.

```
void f(int* &a, int* &b){
  cout << *a << endl;
  a = b;
}

main_program{
  int x=5, y=10;
  int *p = &x, *q = &y;
  f(p,q);
  cout << *p <<' '<< *q << endl;
}
```

What do you think will be printed? Check your answer by executing the code.

9. When drawing letters on the canvas, it may be more convenient if the drawing function can be given a `char` as argument and it then draws out the supplied character. Of course, it may not be possible to draw a character for all values between 0 and 255. So your function should return `true` iff it indeed drew the supplied character.

CHAPTER 10

Recursive Functions

We are now in a position to discuss what is perhaps the most powerful, most versatile problem solving technique ever: recursion. What we are going to present will not really contain any new C++ statements. Rather, what you have learned so far will be used, possibly in a manner which might surprise you, to solve some difficult computational problems in a very succinct manner.

A fundamental idea in designing algorithms is *problem reduction*. The notion is very common in mathematics, where we might say "Using the substitution $y = x^2 + x$ the quartic (fourth degree) equation $(x^2 + x + 5)(x^2 + x + 9) + 7 = 0$ reduces to the quadratic $(y + 5)(y + 9) + 7 = 0$.". Of course, reducing one problem into another is useful only if the new problem is in some sense easier than the original. This is true in our example: quadratic equations are easier to solve than quartic. The strategy of reducing a problem to another is easily expressed in programs: the function we write for solving the first problem will call the function for solving the second problem. We saw examples of this in the previous chapter.

An interesting case of problem reduction is when the new problem is of the *same type* as the original problem. In this case, the reduction is said to be *recursive*. This idea might perhaps appear to be strange, but it is in fact very common. Consider the following rules for differentiation:

$$\frac{d}{dx}(u + v) = \frac{d}{dx}u + \frac{d}{dx}v$$

$$\frac{d}{dx}(uv) = v\frac{d}{dx}u + u\frac{d}{dx}v$$

The first rule, for example, states that the problem of differentiating the sum $u + v$ is the same as that of first differentiating u and v separately, and taking their sum. You have probably used these rules without realizing that they are recursive. There are two reasons why these rules work:

1. The reduced problems are actually simpler in some precise sense. In our example, the problem of differentiating u or of differentiating v are indeed simpler than the problem of differentiating $u + v$, because u and v are both (textually) smaller expressions than $u + v$. Notice that when we reduce one problem to a problem of another type (non-recursive reduction), the new problem is required to be of a simpler type. For recursive reduction, it is enough if the new problem is of a smaller size.

2. We have a way to directly solve problems of some suitably designated small size. Thus we will not just keep reducing problems indefinitely. The problems which we expect to solve directly are called the *base cases*. Considering differentiation again, suppose we wish to compute:

$$\frac{d}{dx}(x \sin x + x)$$

Then using the first rule, we would ask to compute

$$\frac{d}{dx}x \sin x + \frac{d}{dx}x$$

Now, the computation of $\frac{d}{dx}x$ is not done by further reduction, i.e. this is a base case for the procedure. So in this case, we directly write that $\frac{d}{dx}x = 1$. To compute $\frac{d}{dx}x \sin x$ we could use the product rule given above, and we would need to know the base case $\frac{d}{dx}\sin x = \cos x$.

Even on a computer, recursion turns out to be extremely useful. In this chapter we will see several examples of the idea.

10.1 EUCLID'S ALGORITHM FOR GCD

Euclid's algorithm for GCD is naturally expressed recursively, as it turns out. Here is Euclid's theorem, restated for convenience.

Theorem 2 (Euclid) *Let m, n be positive integers. If $m\%n = 0$, then $GCD(m, n) = n$. If $m\%n \neq 0$ then $GCD(m, n) = GCD(n, m\%n)$.*

The theorem essentially says that either the GCD of m, n is n, or it is the GCD of $n, m\%n$. But this is exactly like saying that the derivative of an expression can be written down directly or it is the derivative of some simpler expression. Following the analogy, it would seem tempting, to call `gcd` from inside of itself.

```
int gcd(int m, int n)
// finds GCD(m,n) for positive integers m,n
{
  if(m % n == 0) return n;
  else return gcd(n,m % n);
}
```

A function that calls itself, like the function `gcd` above, is said to be recursive.

Interestingly, `gcd` as defined above actually works! In the last chapter, we sketched out the mechanism used to execute functions, and it turns out that the same mechanism will correctly compute the GCD using the above code. We will see an example and a general proof shortly.

10.1.1 Execution of Recursive `gcd`

Suppose for the moment that our main program that uses the `gcd` definition above is

```
main_program { cout << gcd(36,24) << endl;}
```

Function call	`main_program`	`gcd(36,24)`	`gcd(24,12)`
Activation frame contents		`m` : 36 `n` : 24	`m` : 24 `n` : 12
State of call	Suspended	Suspended	Executing

Code with ▷ showing next statement to be executed	`cout «`
	▷ `gcd(36,24)`
	`« endl;`
		`if(m % n == 0)`
		`  return n;`
		`else return`
		▷ `gcd(n, m % n);`
			▷ `if(m % n == 0)`
			`  return n;`
			`else return`
			`  gcd(n, m % n);`

Fig. 10.1 *A snapshot of the execution of recursive* `gcd`

Suppose the main program begins execution. Immediately, it comes upon the call `gcd(36,24)`. As we know, this causes an activation frame to be constructed for the call, and in this activation frame the parameters `m,n` are assigned the value 36,24 respectively.

Now the execution begins in the new activation frame. The first check, `m % n == 0` fails, because `36 % 24` is 12 and not 0. Thus we execute the `else` part. But this contains the call `gcd(n, m % n)`, i.e. `gcd(24,12)`. Our function call execution mechanism must be used again. Thus, another activation frame is created, this time for `gcd(24,12)`, and `m,n` in this frame are set to 24,12 respectively. Also, the execution of the current call, `gcd(36,24)`, suspends. Figure 10.1 shows the state of the world at this time in the execution.

The execution begins in the new activation frame. We execute the first statement which requires us to check if `m % n == 0`, i.e. `24 % 12 == 0`. This is indeed true. So we execute the statement `return n`. This causes 12 to be returned. Where does this value go? It goes back to the place from where the current recursive call was made. Since the current call was made while processing the second activation frame, the value 12 is returned there. The second activation then continues its execution. However, there isnt much more left to in this activation. This code was to return the value of `gcd(24,12)`—now that this value is known, 12, it is returned back. So the value 12 is returned also from the second activation. This goes back to the first activation. The first activation resumes from where it was suspended. As per its code, it prints out the received value, 12, and then `main` terminates.

So as you can see, the correct value was computed and printed.

The number of times a function is called is called the *depth* of the recursion. In the present example, the depth is 2. For the example considered in Section 7.7, $GCD(3977, 943)$ you can check that the depth will be 5, equalling the number of iterations required by the program in Section 7.7. Please work this out by hand. You will also observe that the values assigned to `m,n` in successive activation frames in the recursive program are in fact the same values that `m,n` receive in successive iterations of the non-recursive program in Section 7.7.

10.1.2 Interpretation of Recursive Programs

In some ways there is nothing more to be said about recursive programs—the last section said it all. We mentioned in the previous chapter that a function call should be thought of as contracting an agent to do the work you want, while you wait (are suspended). If the work taken up by the agent is too complicated, then it is possible that the agent might further subcontract it out to another (sub-)agent. When this happens, you are waiting for the agent to finish, the agent is herself waiting for the sub-agent to finish, and possibly the chain might be very long. But so what?

Of course, the natural intuition is that you contract out work that you cannot do yourself. So it makes sense for the function `lcm` to contract out the work of `gcd` as was done in the last chapter. But whoever heard of contracting out work that you yourself can do? That is in fact what seems to be happening: the recursive function `gcd` clearly should know how to compute the GCD, and yet it seems to be subcontracting out work!

Suppose you have the task of building a Russian doll, which is a children's toy which looks like a doll, but you can open up the doll to see that there is a doll inside, which contains another doll, and so on, till some fairly small doll is reached, which cannot be further opened up. Suppose further that we define a k-level doll to be a doll which contains $k-1$ dolls inside. So let us say that your task is of building a k-level doll. How would you do it recursively?

You would contract it to some craftsman. Imagine that the craftsman builds the outer doll, but does not work on the inner dolls. Instead the task of building the inner $k-1$ dolls, which is really a $k-1$ level doll is subcontracted to another craftsman. And so it goes. This continues until a craftsman is asked to build a 0 level doll, which is just an ordinary doll. This is not subcontracted but built directly. So this doll is sent back to the previous craftsman who adds a doll and makes it a level 1 doll and sends it back, and so on, until you eventually receive the k level doll that you ordered!

This is clearly a strange way of building dolls—but what you should understand for now is that it can work in principle. To prove that the process works correctly, you would use induction. First establish that some craftsman can build a level 0 doll without further subcontracting, the so called *base case*. Next, you must prove that a craftsman can put together a level $i+1$ doll given a level i doll. This is the inductive step.

We see how this works for GCD next.

10.1.3 Correctness of Recursive Programs

The key to proving the correctness of recursive programs is to use mathematical induction. We first need to have a notion of the *size of the problem being solved*. The induction hypothesis typically states that the program correctly solves problems of a certain size.

As an example we will see how to argue that our `gcd` function will correctly compute the GCD. We need to decide what should constiute the size. Based on the experience from Section 7.7, we choose the value of the second argument as the size of the problem. Our induction hypothesis $IH(j)$ is: `gcd(`i, j`)` correctly computes the GCD of i, j for all $i > 0$.

It is worth taking an example to clarify this. For example, $IH(10)$ is: `gcd(`$i, 10$`)` correctly computes the GCD of $i, 10$ for all $i > 0$. Clearly, if we prove $IH(j)$ for all $j > 0$ then we will have proved that `gcd(`i, j`)` works correctly for all $i, j > 0$.

The base case is $j = 1$. Thus, we must prove that any call `gcd(`$i, 1$`)` will return the correct result. But note that in the first step of `gcd(`$i, 1$`)` we will discover that $i\%1$ is zero, and will report 1 as the answer. This is clearly correct.

So let us assume $IH(1), IH(2), \ldots, IH(j)$. Using these, we will prove $IH(j+1)$. In other words, we must prove that the call `gcd(`$i, j+1$`)` produces the correct answer. In the first step of the call, we check whether $i\%(j+1)$ equals 0. This is true if $j+1$ divides i, and in that case $j+1$ is the GCD, which is correctly returned. Suppose then that the condition is false. In that case the algorithm tries to compute and return `gcd(`$j, i\%(j+1)$`)`, where we know that $(i\%(j+1)) > 0$. But the remainder on dividing by $j+1$ must be at most j. Thus, the second argument to this call lies between 1 and j. Thus, by one of our assumptions $IH(1), \ldots, IH(j)$, we know that this call will return correctly, i.e. return

$GCD(j, i\%(j+1))$. But by Euclid's theorem, we know that this is also $GCD(i, j+1)$, i.e. it is the correct answer. Thus, correctness follows by the principle of (strong) induction.

You will observe that this proof is very similar to the proof in Section 7.8. Indeed, the non-recursive program in that section is really doing the same calculations as the recursive one: the values of `m, n` in the tth iteration will be the same as the values taken by the arguments `m, n` in the tth recursion. Indeed, other assertion made there, e.g. that the number of iterations must be $O(\log m)$, if $GCD(m, n)$ is being calculated, will correspond to the assertion in the present case that the recursive program will recurse only to a depth $O(\log m)$. Thus the total time taken will be $O(\log m)$ as for the non-recursive algorithm.

10.2 RECURSIVE PICTURES

Figure 10.2 shows a picture which we might consider, using some imagination, to be of a tree, say from the African Savannas. Our goal in this section is to write a program to draw such trees. Note that our interest in trees goes beyond botany; tree diagrams are used in many places. Thus, a tree might depict the hierarchical structure of many organizations, e.g. the root might represent the president, and that may be connected to the vice presidents who report to the president, those in turn to the managers who report to the vice presidents. As you will see later, the manner in which functions are called also have a tree structure. So understanding tree structures and being able to draw them is useful. By the way, such tree structures are studied at great length in *graph theory*, where it is customary to use the term *edge* for a branch of a tree, and the term vertex for each endpoint (possibly shared) of each branch.

Figure 10.2 has some interesting symmetries. First, of course, there is a symmetry of reflection about a vertical line through the middle. But also to be noted is the another kind of symmetry: parts of the tree are similar to the whole. The portion of the tree on top of the left branch from the bottom, or the portion on top of the right branch, can each be considered to be a tree! In fact we might describe a tree as two small trees on top of a "V" shape formed by the branches at the bottom.

It is customary to define the *height* of the tree to be the maximum number of branches you travel over as you move up from the root towards the top along any path. Our tree of Figure 10.2 has height 5. Clearly, we can say that a tree of height h is made up of a *root* with two branches going out, on top of each of which sits a height $h-1$ tree, as shown in Figure 10.3. Of course, to draw the picture, we need more information, for example, what is the length of the branches, and what are the angles at which the branches emerge. For the tree shown, the branch lengths shrink as we go upwards, and so do the angles. Suppose we declare that we want both the branch lengths and the angles between emerging

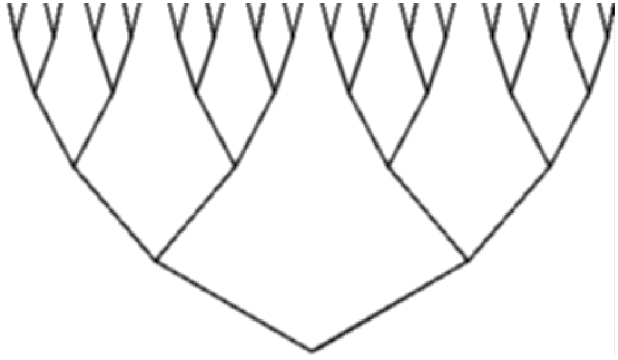


Fig. 10.2 *An exotic tree*

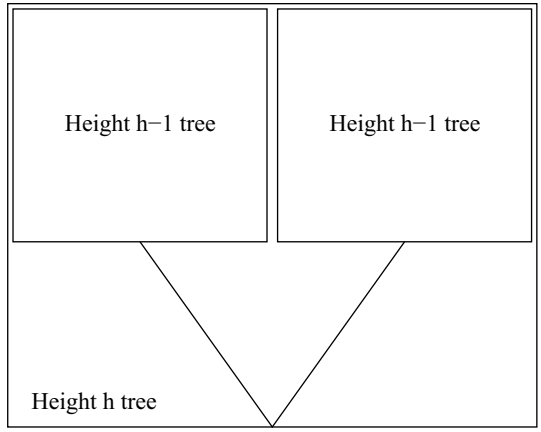


Fig. 10.3 *Recursive structure of our exotic tree*

branches to both shrink by a fixed *shrinkage* factor as we go up. Now, if we are given the length of the bottom most branches, and the height of the tree, we should be able to draw the picture.

The code follows the basic observation: to draw a tree of height $h > 0$, we must draw the root and immediate branches, and two trees of height $h - 1$ on top. A tree of height $h = 0$ is just a point, and so nothing need be drawn.

10.2.1 Drawing the Tree Using a Turtle

As in any turtle-based drawing program, we must carefully write the pre and post conditions for the turtle. So we will require that at the beginning the turtle be at the root, pointing upwards (pre condition). After the drawing is finished, we will ensure that the turtle is again at the root and pointing upwards (post condition). Once we fix these pre and post conditions, the program writes itself: we merely have to ensure that we maintain the conditions.

The process of drawing is as follows. Clearly, if $h = 0$, we draw nothing and return. Otherwise, the figure is drawn in a series of 7 steps.

1. Draw the left branch. At the start, because of the precondition, we know that the turtle is pointing upwards, so it must turn by half the angle that is meant to be between the branches. Then we move forward by the length of the branch.
2. Draw the left subtree. We first turn so that the turtle is facing the top direction, because that is a precondition for drawing trees. Then we recurse. We need to call with height $h - 1$, and the branch length and angle shrunk by the given shrinkage value.
3. Go back to the root. After drawing the left subtree, its post-condition guarantees that the turtle will face directly upwards. To get back to the root we must turn and go backwards, which is accomplished by giving a negative argument to the `forward` command.
4. Draw the right branch. Since the turtle is pointing in the direction of the left branch, we must turn it to the right, and then go forward.
5. Draw the right subtree. This is exactly as we did the left.
6. Go back to the root, as we did after drawing the left subtree.

7. Ensure the post condition. Finally, we want to honour the post-condition, so we turn the turtle so that it faces directly upward.

This is expressed as the following program, where we have put comments to indicate the correspondence with the steps described above.

```
void tree(int height, float length, float angle, float shrinkage)
  // precondition: turtle is at root, pointing up.
  // post condition: same
{
  if(height == 0) return;
                          // 1.  draw the left branch
  left(angle/2);
  forward(length);
                         // 2.  draw the left (sub)tree.
  right(angle/2);
  tree(height-1, length*shrinkage, angle*shrinkage, shrinkage);

                         // 3.  go back to the root
  left(angle/2);
  forward(-length);
                         // 4.  draw the right branch
  right(angle);
  forward(length);
                         // 5.  draw right (sub)tree.
  left(angle/2);
  tree(height-1,length*shrinkage, angle*shrinkage, shrinkage);
                         // 6.  go back to root
  right(angle/2);
  forward(-length);
                         // 7.  ensure post condition
  left(angle/2);
}
```

To call the function, we must supply the arguments, but also ensure the precondition. Since we know that at the start the turtle is facing right, we must turn it left by 90 degrees so that it points upwards. So our main program could be the following.

```
main_program{
  turtlesim();
  left(90);

  tree(5,120,120,0.68);

  wait(5);
}
```

10.2.2 Trees Without Using a Turtle

It is easier to draw a tree without using a turtle, as we will show next. However, using a turtle has some advantages, which we remark upon at the end.

The basic idea is to use the `Line` shape from Chapter 5. We draw the branches using this, and then recurse to draw the subtrees. Note that we must now pass the coordinates of the root as well to each call. For variety, we will also draw a tree with a somewhat different layout. Specifically, we will draw a tree such that it occupies a given rectangular box, with the root appearing at the center of its base. If the coordinates of the root are given, then the box is specified by giving its height H_b and width W_b. We will also assume for simplicity that the for a tree of height h the points at which the branches divide are at heights $H_b/h, 2H_b/h, 3H_b/h, \ldots$. Likewise, when a tree has two subtrees, each subtree is accommodated in a box of half the width of the original box. The code can now be written easily.

```
void tree(int h, float H_b, float W_b,
   float rx, float ry) // coordinates of the root.
{
  if(h > 0){
    float LSRx = rx-W_b/4; // x coordinate of root of Left subtree.
    float RSRx = rx+W_b/4; // x coordinate of root of Right subtree.
    float SRy  = ry-H_b/h; // y coordinate of roots of subtrees.


    Line Lbranch(rx, ry, LSRx, SRy); Lbranch.imprint();
    Line Rbranch(rx, ry, RSRx, SRy); Rbranch.imprint();

    tree(h-1, H_b-H_b/h, W_b/2, LSRx, SRy); //Left Subtree.
    tree(h-1, H_b-H_b/h, W_b/2, RSRx, SRy); //Right Subtree.
  }
}
```

This code is more compact, because we don't have to worry about managing the postconditions of the turtle.

However, it should be noted that this code is only useful to grow trees vertically. Suppose you want to orient the tree at an angle of 60 degrees to the vertical, then this code is useless. However, the turtle-based code can be used, we merely call it after the turtle is oriented at the required angle. This feature appears useful for drawing many botanical trees, i.e. the subtrees of many botanical trees appear grow at an angle to the vertical. As a result, to draw realistic looking (botanical) trees, it might be more convenient to use the turtle based code. See Exercise 11.

10.3 VIRAHANKA NUMBERS

Virahanka, an Indian prosodist who lived in the 7th century AD considered the following problem. Suppose you want to construct poetic metres, built using syllables of length 1 and length 2. The length of a poetic metre is simply the sum of the lengths of the syllables in the metre. Virahanka asked and answered the following question: how many different poetic metres can you compose of a given length?

You may find the problem easier to understand if expressed in a slightly different form. Suppose you have an unlimited supply of bricks of heights 1 and 2. You want to construct a tower of height n. In how many ways can you do this? For example, suppose $n = 4$. One possibility is to stack 4 height 1 bricks, i.e. the order of heights considered bottom to top is 1,1,1,1. Other orders are 1,1,2, or 1,2,1 or 2,1,1 or 2,2. You can check by trial and error that no other orders are possible. Thus if you define V_n to be the number of ways in which a tower of height n can be constructed, we have demonstrated that $V_4 = 5$. We would like to write a program that computes V_n given n. Clearly, V_n is also the number of different poetic metres of total length n composed using syllables of lengths 1 and 2.

Virahanka, or quite possibly preceding prosodists, about whom definite information is not known, used the following method to solve the problem, which apparently originates from Pingala, who lived around the 3rd century BCE. Virahanka begins with the observation that the first syllable must have a length of 1 or 2, or that the bottom-most brick is either of height 1 or height 2. You may think this is rather obvious, but from this it follows:

$$V_n = \text{Number of ways of building a tower of height } n = \text{Number of ways of building a tower of height } n \text{ with bottom-most brick of height 1} + \text{Number of ways of building a tower of height } n \text{ with bottom-most brick of height 2.}$$

Virahanka's next observation, in the language of bricks, is that if you select the bottom-most brick to be of height 1, then the problem of building the rest of the tower is simply the problem of building a height $n - 1$ tower. Thus, we have

$$\text{Number of ways of building a tower of height } n \text{ with bottom-most brick of size 1} = \text{Number of ways of building a tower of height } n - 1 = V_{n-1}$$

Likewise, it also follows that

$$\text{Number of ways of building a tower of height } n \text{ with bottom-most brick of height 2} = \text{Number of ways of building a tower of height } n - 2 = V_{n-2}$$

So we have

$$V_n = V_{n-1} + V_{n-2}$$

What we have written above is an example of a *recurrence*, an equation which recursively defines a sequence of numbers, $V_1, V_2, \ldots$ in our case.

Now we are ready to write a recursive program. Clearly, in order to solve the problem of size n, we need a solution to problems of size $n - 1$ and $n - 2$ respectively. So we have a procedure for reducing the size of the problem, what we need is the base case. Is there a problem that we can solve easily? Clearly, $V_1 = 1$, because a height 1 tower can be built in only 1 way – by using a single height 1 brick. So we may propose the following program:

```
// First attempt
int Virahanka(int n){
  if(n == 1) return 1; // V_1
  else return Virahanka(n-1) + Virahanka(n-2);
}
```

If we run this, say calling it `Virahanka(10)` from a suitable main program, we will see that it does not terminate. How do we figure out what is wrong? We could try to prove the correctness of our function: this will clearly not succeed, but it may reveal our error.

Clearly, the value of the argument to `Virahanka` should be considered the "problem size". So our induction hypothesis $IH(i)$ is: `Virahanka(`i`)` returns V_i for $i \geq 1$.

The base case is $i = 1$. When we make the call `Virahanka(1)`, in the very first line we will check if `n` is 1. This will turn out to be true, because `n` has the value $i = 1$. So 1 will be returned. Thus, the base case is proved.

So let us now assume $IH(1), IH(2), \ldots, IH(i)$. We would like to prove $IH(i+1)$, i.e. show that V_{i+1} is returned for the call `Virahanka(`$i+1$`)`, where $i \geq 2$. On such a call, clearly the test in the first statement will not succeed, so we will return `Virahanka(n-1) + Virahanka(n-2)`. Since `n` is $i+1$, these calls are to `Virahanka(`i`)` and `Virahanka(`$i-1$`)`. We know from the assumption that `Virahanka` called with an argument between 1 and i will return V_i. Further, since $i \geq 2$, we know that $i - 1 \geq 1$. Thus, `Virahanka(`$i-1$`)` will indeed return V_{i-1}. As to the call, `Virahanka(`$i-2$`)`, we cannot be sure, because we only know that $i - 2 \geq 0$. Thus $i - 2$ may not always lie in the range 1 through i. Specifically, if $i = 2$, then `Virahanka` is called with 0. Our assumptions do not cover this case $IH(0)$, and the proof cannot be completed.

But our failed proof tells us something: our function calls `Virahanka(0)` recursively when we make the call `Virahanka(2)`. We do not expect `Virahanka` to be called with argument 0. So we should really try to rework what happens in this case.

The simple fix is: we must handle `Virahanka(2)` also in a base case. We know that Virahanka(2) should return V_2 which is 2.

```
int Virahanka(int n){
  if(n == 1) return 1; // V_1
  if(n == 2) return 2; // V_2
  return Virahanka(n-1) + Virahanka(n-2); // V_{n-1} + V_{n-2}
}
```

Indeed you will see that this program will run correctly, and that our proof will also go through. This is left as an exercise.

In hindsight, we could have expected this: just knowing $V_1 = 1$ and $V_n = V_{n-1} + V_{n-2}$, we cannot figure out V_2. So we need to specify that.

10.3.1 Some Difficulties

If you run this, you will see that it is very slow, even for modestly large n. The reason for it can be seen in Figure 10.4. This figure shows the so called *execution tree* for the call `Virahanka(6)`. Note that we have drawn this tree growing downward, as is more customary for execution trees. The node at the top, which we will still call the root, corresponds to the original call. So we have marked the root in the picture with 6, the argument to the original call. Out of each vertex we have one downward going edge for every call made. Since `Virahanka(6)` requires `Virahanka` to be called first with argument 5, and then with 4, we have 2 outgoing branches. At the ends of the corresponding branches we have put down 5 and 4 respectively, the arguments for the corresponding calls. This goes on till we get to calls `Virahanka(2)` or `Virahanka(1)`. Since these calls do not make further recursive calls, there are no outgoing branches from the vertices corresponding to these calls.

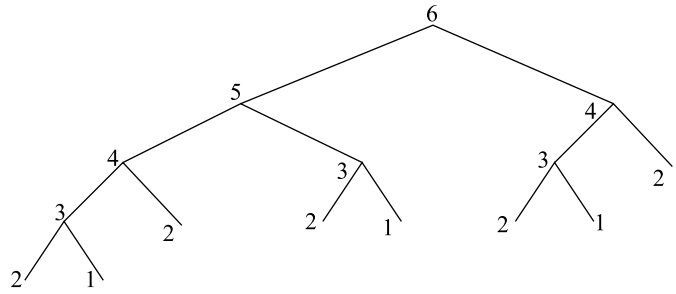


Fig. 10.4 *Execution tree for* `Virahanka(6)`

As you can see in the figure, `Virahanka(4)` is called twice, once as a part of `Virahanka(5)`, and once directly from `Virahanka(6)`. But once we know V_4 through one of the calls, we could just remember this value and use it instead of making the second call to `Virahanka(4)`. In fact, you will see that the call `Virahanka(3)` happens 3 times, the call `Virahanka(2)` happens 5 times, and the call `Virahanka(1)` happens 8 times. So the program is quite wasteful. You can in fact prove that while computing V_n, our function will make V_{n-i} calls to `Virahanka(`i`)`. The number of calls to `Virahanka(1)` is thus V_{n-1}, which you can show to be at least $2^{\lfloor n/2 \rfloor}$. Thus, if you want to compute V_n by using our function above you are expecting to spend time proportional to at least $2^{n/2}$. This is a huge number, and indeed computing something like say V_{45} using the call `Virahanka(45)` takes an enormous amount of time on most computers.

10.3.2 Using a Loop

Here is a different way to compute V_n. We know that $V_1 = 1$, $V_2 = 2$ and $V_3 = V_1 + V_2$. Thus, we can compute $V_3 = V_2 + V_1 = 2 + 1 = 3$. After that we can compute $V_4 = V_3 + V_2 = 3 + 2 = 5$. After that we can compute V_5, and this process can go on. Clearly, there is something repetitive going on here. So presumably a loop will be useful to program it. Presumably, we can compute one Virahanka number in each iteration, and there need to be $n - 2$ iterations, because V_1, V_2 were computed before entering the loop.

Our plan then is to calculate V_{i+2} at the end of the ith iteration, so that at the end of the $n - 2$th iteration we will have V_n. At the beginning of the ith iteration we should have available V_i and V_{i+1}. For storing these, we will use variables `vi` and `viplus1`. During the `i`th iteration we will calculate V_{i+2} in variable `viplus2`. Note that to prepare for the next iteration, we must set `vi` to `viplus1` and `viplus1` to `viplus2`. And we must initialize `vi, viplus1` respectively to V_1, V_2.

```
int VirahankaByLooping(int n){    // Program to compute Virahanka
                                  // number V_n
  int v1=1,v2=2;                  // v1 = V_1,  v2 = V_2
  if(n == 1) return v1;
  else if(n == 2) return v2;
  else {
    int vi=v1, viplus1=v2, viplus2;
    repeat(n-2){// Plan:
          // Before ith iteration, vi, viplus1 = V_{i}, V_{i+1}
      viplus2 = vi + viplus1;
```

```
      vi = viplus1;              // prepare for next iteration
      viplus1 = viplus2;
    }
    return viplus2;
  }
}
```

To compute V_n, this function will require $n-2$ iterations. Each iteration takes a fixed amount of time independent of n. Thus, we can say that the total time is approximately proportional to n.

Indeed, you will see that V_{45} gets computed essentially instantaneously using this loop based function.

10.3.3 Historical Remarks

This sequence may look familiar to many readers. Indeed, these numbers are more commonly known as the Fibonacci numbers. But Virahanka is known to have studied them before Fibonacci. In fact, it appears that they may have been known in India even before Virahanka. In any case, it seems more appropriate to call these numbers Virahanka numbers rather than Fibonacci numbers.

10.4 THE GAME OF NIM

In this section, we will write a program to play the game of Nim. This game is quite simple, but nevertheless interesting, and our program will contain a key idea which will be useful in all game playing programs.

The game has two players, say White and Black. There are some n piles of stones, the ith pile containing x_i stones at the beginning. We will have different games for different choices of x_i. A move for a player involves the player picking a pile in which there is at least one stone, and removing one or more stones from that pile. The players move alternately, say with White moving first. The player that makes the last valid move, i.e. after which no stones are left, is considered the winner. Or you may say that the player who is unable to make a move on his turn because there are no stones left is the loser.

Here is a simple example. Suppose we have only 2 piles initially with 5 and 3 stones respectively. Suppose White picks 4 stones from pile 1. Then the first pile has 1 stone left and the second has 3. Now Black can win by picking 2 from the second pile: this will leave 1 stone in each pile, and then White can pick only 1 of them, leaving the last one for Black. Of course, White need not have picked 4 stones in the very first move. Is there a different choice for which he can ensure a win? We will leave it to you to observe that White can in fact win this game by picking only 2 stones from the first pile in his first move.

So here is the central question of this section: Given a game position, i.e. number of stones in each pile, determine whether the player whose turn it is to play can win, no matter what the other player plays. In case the position is winning, we would also like to determine a winning move. Note by the way, that when we say winning/losing position, we mean winning/losing for the player whose turn it is to move.

In trying to solve any problem, it is a good idea to try out some examples first. Consider the simplest possible position: the position in which no pile contains any stone because all were taken earlier. As defined above, in this position, the player whose turn it is is clearly the loser. The next harder example

is: suppose there is only one pile with just one stone. Clearly, this is a winning position (for the player on move) because he can take that stone. In fact, if there is just one pile with any number of stones, it is a winning position because the player on move will take the entire pile.

Let us next consider the next more complex situation, say there are 2 piles, each with one stone. There are only two (similar) moves possible, either take the stone in the first pile, or the stone in the second pile. In either case, we leave behind a single pile with 1 stone, which is a winning position for our opponent. Thus, an important principle emerges from this example:

❑ *Observation 1*

If from a certain position P, suppose on every move we go to a winning position. Then P is a losing position.

This is indeed an important observation. Let us keep going and consider more complex situations, say there are two piles, in the first one there are 2 stones, and the second has only 1. Now we have a choice of three moves:

1. Pick one stone from the first pile. In that case, one stone remains in each pile. As we have discussed, this is a lost position for our opponent. So good for us!
2. Pick two stones from the first pile. In this case, we leave just one stone. This is a winning position, for our opponent. Hence, not good for us.
3. Pick the only stone from the second pile. This leaves behind one pile with two stones. As discussed above, this is also a winning position, for our opponent. Hence, this is not good either.

So in this position, the first move will make us win while the remaining two will make us lose. So what do we make of this position? Remember that *we* have the choice of what move to make, and hence we will certainly choose the first move! So this position is a winning position for us. This seems to generalize into another observation.

❑ *Observation 2*

If in some position P there exists a move after which we reach a losing position. Then P is a winning position.

Note that this observation nicely complements the first one. If we find that some move leads to a losing position, then the second observation applies. If we find that there is no such move, i.e. all moves lead to a winning position, then the first observation applies.

The above observations gives us a recursive algorithm for determining if a given position P is winning or losing. We determine all moves m_i possible in P, and the positions P_i they lead to. Then we determine (recursively!) whether P_i are winning or losing. If we find some P_i that is losing, we declare P to be winning. Otherwise if all P_i are winning, we declare P to be losing. We know that in order for a recursive algorithm to work, we must ensure two things:

1. Each subproblem we are required to solve is simpler than the original. In our case, this is true in the sense that each P_i must have at least one stone less than P, and is hence simpler.
2. We can argue that eventually we will reach some ("simplest") problems which we can solve directly. Clearly, as the game progresses we will reach the situation in which no stone remains. As discussed, this is a losing position.

Thus, we can write a program to determine whether a Nim position is winning or losing. We give this program for the case of 3 piles, but you can see that it can be easily extended for a larger number of piles. The function `winning` given below takes a game position and returns `true` if the position is winning, and `false` if the position is losing.

```
bool winning(int x, int y, int z)
// x,y,z = number of stones in the 3 piles.
// returns true if this is a winning position.
{
  if(x==0 && y==0 && z==0) return false; // base case

  for(int i=1; i<=x; ++i)      // Pick i stones from pile 1
    if (!winning(x-i,y,z)) return true;
                               // if a losing next state is found

  for(int i=1; i<=y; ++i)      // Pick i stones from pile 2
    if (!winning(x,y-i,z)) return true;
                               // if a losing next state is found

  for(int i=1; i<=z; ++i)      // Pick i stones from pile 3
    if (!winning(x,y,z-i)) return true;
                               // if a losing next state is found

  return false;                // if all next states are winning
}
```

The function can be called using a main program such as the one below.

```
main_program{
  int x,y,z;
  cout << "Give the number of stones in the 3 piles: ";
  cin >> x >> y >> z;
  if (winning(x,y,z)) cout << "Wins." << endl;
  else cout << "Loses." <<endl;
}
```

Our function only says whether the given position is winning or losing, it does not say what move to play if it is a winning position. You can easily modify the program to do this, as you are asked in the exercises.

The logic of our function should be clear. In the given position, we can choose to take stones from either the first pile, the second pile, or the third pile. The first loop in the function considers in turn the case in which we remove `i` stones from the first pile. This leaves the new position in which the number of stones is `x-i,y,z`. We recursively check if this is a losing position. If so, the original position, i.e. in which there are `x,y,z` stones respectively, must be a winning position by observation 2. The subsequent loops consider the cases in which we remove stones from the second and third piles. If we find no losing position after checking all moves, then indeed the original position must be losing, by observation 1. So we return `false` in the last statement of the function.

10.4.1 Remarks

It turns out that there is a more direct way to say whether a given position is winning or losing. This is very clever, involving writing the number of stones in the piles in binary, and performing additions of the bits modulo 2, and so on. We will not cover this, but you should be able to find it discussed on the Internet.

Our program is nevertheless interesting, because the general structure of the program applies for many 2 player games. Indeed, recursion is an important tool in writing game playing programs.

10.5 CONCLUDING REMARKS

Recursion is an extremely important idea for designing algorithms. Suppose you want to solve an instance of a certain problem, i.e. a certain problem for a given set of input values. Then it helps to assume that you can solve smaller instances (same problem for a smaller set of input values) somehow, and ask: will the solution of smaller instances help me in solving the larger instance. If you can relate the solution of your instance to solution of smaller instances, and also find a way to solve small instances ("base cases"), you have a recursive algorithm! It is possible that Euclid discovered his GCD algorithm thinking in this manner. Virahanka probably also discovered the solution to his problem thinking in this manner.

In addition to having recursive algorithms, we also have recursive structures. Trees are a good example of objects with a recursive structures. We can exploit the recursive structure to design a recursive algorithm for drawing trees. But as you will see later, trees will be used for representing many familiar objects, and in fact designing algorithms for those objects will require exploiting the recursive structure.

The notion of recurrences is also important.

The example of Virahanka numbers also demonstrates an interesting point. It is quite likely that Virahanka also solved his problem by thinking recursively, and indeed it is a good idea to think recursively for the purpose of solving problems. But we must remember that sometimes it may not be best to write the program recursively.

A technical point should be noted regarding the calculation of Virahanka numbers. As you will show in the exercises, V_n is at least $2^{n/2}$, and hence even for modest value of n it will not fit in an `int`. If you use `long long` you can work with somewhat larger values. If you use `double` you can work with much larger values of n, but they will be correct only to 15 digits or so.

Finally, we also saw how to prove the correctness of recursive functions. We also saw how trying to prove the correctness of an incorrect program can help us locate the error in the program.

EXERCISES

1. The factorial of a number n is denoted as $n!$, and can be defined using the recurrence $n! = n \times (n-1)!$ for $n > 0$ and $0! = 1$. Write a recursive function to compute $n!$.
2. The binomial coefficient $\binom{n}{r}$ can be defined recursively as $\binom{n}{r} = \binom{n-1}{r} + \binom{n-1}{r-1}$, for $n > r > 0$ and $\binom{n}{0} = \binom{n}{n} = 1$ for all $n \geq 0$. Write a function to compute $\binom{n}{r}$.
3. Consider an equation $ax + by = c$, where a, b, c are integers, and the unknowns x, y are required to be integers. Such equations are called Diaphontine equations. If $GCD(a, b)$ does not divide

c, then the equation does not have any solution. However, the equation will have infinitely many solutions if $GCD(a, b)$ does divide c. Write a program which takes a, b, c as input and prints a solution if $GCD(a, b)$ divides c.

Hint 1: Suppose $a = 1$. Show that in this case an integer solution is easily obtained.

Hint 2: Suppose the equation is $17x + 10y = 4$. Suppose you substitute $y = z - x$. Then you get the new equation $7x + 10z = 4$. Observe that the new equation has smaller coefficients, and given a solution to the new equation you can get a solution to the old one.

4. Consider the recurrence $W_n = W_{n-1} + W_{n-2} + W_{n-3}$, with $W_0 = W_1 = W_2 = 1$. Write a recursive program for printing W_n. Also write a loop based program.

5. Let B_n denote the number of branches in the recursion tree for V_n. Thus $B_6 = 14$, considering Figure 10.4. Note that each branch ends in a call to `Virahanka`, hence B_n gives a good estimate of the number of operations needed to compute V_n. Write a recurrence for B_n and use it to write a program that computes B_n. What are the base cases for this? Make sure your answer matches the branches in the trees of Figure 10.4.

6. Suppose you call the function `gcd` on consecutive Virahanka numbers V_n, V_{n+1}, i.e. you execute `gcd(`V_{n+1}, V_n`)`. There is something interesting about the arguments to the successive recursive calls. What is it? You may wish to modify `gcd` so that it prints the arguments, and see what results. The *depth* of the recursion is defined to be the number of consecutive recursive calls made, each nested inside the preceding one. What is the depth of the recursion for the call `gcd(`V_{n+1}, V_n`)`?

7. The tree drawn in Figure 10.2 is called a complete binary tree. Binary, because at each branching point there are exactly 2 branches, or at the top, where they are no branches. Complete, because all branches go to the same height. You could have an incomplete binary tree also, say you only have one branch on one side and the entire tree on the other.

Write a program which takes inputs from the user and draws any binary tree. Suppose to any request the user will only answer 0 (false) or (true). Device a system of questions using which you can determine how to move the turtle. Make sure you ask as few questions as possible.

8. Consider a complete binary tree with height h. As you can see, such a tree has $2^{h+1} - 1$ vertices. Our goal now is to write a program that not only draws such a tree, but also assigns a unique number for each vertex, in the range 1 through $2^{h+1} - 1$. Further, the number should be printed in the picture, slightly to the right of the vertex itself. The simplest numbering is the *In-order* numbering. In this, the vertices are numbered 1 through $2^{h+1} - 1$ in left to right order. Thus the leftmost leaf would get the number 1, the root of the entire tree would get the number 2^h, and the rightmost leaf would get the number $2^{h+1} - 1$. Modify our program drawing trees without using the turtle so that this numbering is also printed. *Hint:* Have extra arguments to the call which tell the range of numbers to be printed in that call. Another possibility is to have a reference argument which states what number to print. It should start at 0 and must be incremented everytime a number is printed.

9. Another interesting numbering of tree nodes is the *pre-order* numbering. The pre-order time of a node is simply the time at which the subtree rooted at that node starts getting drawn. Based on this, the pre-order number of a node is defined to be i if its preorder time is the ith smallest.

Modify our tree drawing program (not using turtle) so that it prints the pre-order numbers along with the tree. As examples, note that the root has pre-order number 1, the leftmost leaf the number $h + 1$, and the rightmost leaf the number $2^{h+1} - 1$.

10. Another interesting numbering of tree nodes is the *post-order* numbering. The post-order time of a node is simply the time at which drawing of the subtree rooted at that node is finished. Based on this, the post-order number of a node is defined to be i if its post-order time is the ith smallest. Modify our tree-drawing program (not using turtle) so that it prints the post-order numbers along with the tree. As examples, note that the leftmost leaf has post-order number 1, the root the number $2^{h+1} - 1$, and the rightmost leaf the number $2^{h+1} - h - 1$.

11. More commonly, (botanical) trees have a single trunk that rises vertically, and then splits into branches. So you could consider a tree to be "one vertical branch, with two trees growing out of it at an angle". Draw trees expressing this idea as a recursive program. It will be convenient to use the turtle for this. Try out variations, find which trees look realistic. Figure 10.5 shows an example, taken from the literature on the Logo programming language. The triangles at the end are obtained by imprinting the turtle.

Fig. 10.5 *A realistic tree*

12. Write a function `draw_Hem` that draws the recursion tree for calls to `Virahanka`, i.e. `draw_Hem(6)` should be able to construct the tree shown in Figure 10.4.

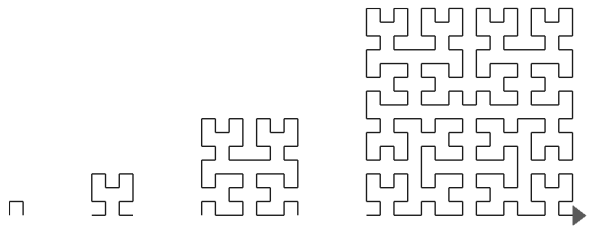


Fig. 10.6 *Hilbert space filling curves* H_1, H_2, H_3, H_4

13. Figure 10.6 shows curves H_1, H_2, H_3 and H_4, left to right. These curves were invented by the mathematician David Hilbert, and are examples of so-called *space-filling* curves.
Draw a sketch (on paper) showing how H_n is composed of one or more H_{n-1} curves. This should be in the style of Figure 10.3. Write a turtle-based program to draw a Hilbert space filling curve H_n given n.
Follow the general scheme we used in Section 10.2, i.e. begin by stating the pre and post conditions for the turtle for drawing H_n. Try to draw the curve without lifting the pen or overdrawing.
You may find the following fact useful. Suppose a certain function `f` draws some figure `F`. Then if we replace every turning angle θ in `f` by $-\theta$, then we will get a figure that is a mirror image of `F`.

14. Write a recursive function for finding x^n where n is an integer. Try to get an algorithm which requires far fewer than the $n-1$ multiplications needed in the natural algorithm of multiplying x with itself. *Hint:* Suppose n is even. Then observe that you can calculate x^n by (recursively) calculating $x^{n/2}$ and then squaring it, i.e. computing $x^{n/2} \times x^{n/2}$. This will enable you to compute, say, x^{16} using just 4 multiplications. The question you must answer is: what do you do if n is odd. Try something relatively simple.

15. There are many variations possible on the game of Nim as described above. One variation is: the player who moves last loses. How will you determine whether a position is winning or losing for this new game?

16. In another variation, you are allowed to pick either any non-zero number of stones from a single pile, or an equal number of stones from two piles. Write a function that says whether a position is winning for this game.

17. Write a function which returns a 0 if the given position is losing, but if the position is winning, returns a value that somehow indicates what move to make. Decide on a suitable encoding to do this. For example, to indicate that s stones should be picked from pile p, return the number $p \times 10^m + s$, where 10^m is a power of 10 larger than x, y, z the number of stones in the piles for the given position. With this encoding, the last m digits will give the number of stones to pick, and the more significant digits will indicate the pile number. Another possibility is to use reference arguments to return how many stones to move and from which pile.

18. Suppose you want to send a message using the following very simple code. Say your message only consists of the letters 'a' through 'z', and in the code your merely replace the ith letter by $i-1$. Thus, 'a' will be coded by 0, 'b' by 1, and so on till 'z' by 25. Further, there are no separators between the numbers corresponding to each letter. Thus, the word "bat" will be coded as the string 1019. Clearly, some strings will have more than one interpretation, i.e. the string 1019 could also have come from "kt". You are to write a program that takes such a sequence of numbers, reads it one character at a time and prints out the number of ways in which it can be interpreted.
Hint: This is somewhat tricky. Let N_i denote the number of ways in which the first i characters can be interpreted. Show that you can determine N_i knowing N_{i-1} and N_{i-2}, and the ith and $i-1$th characters. Now you should be able to write an iterative program to determine N_i.

CHAPTER 11

Program Organization and Functions

We discussed functions as a way of encapsulating frequently used code so that it can be written just once and then called whenever needed. However, functions also play another role in C++ programs. Just as a cell is a basic structural unit of life, a function can be considered to be an organizational unit of a C++ programs. A C++ program essentially is a collection of functions[1]. As we will see shortly, even the `main_program` you have been writing is turned into a function by Simplecpp, structurally similar to the other functions we have been writing. Managing the functions constituting a program, especially large programs, requires a systematic approach. The problem is further complicated because large programs are often developed by teams of programmers, with each programmer possibly responsible for developing some of the functions. Clearly, it is convenient if the different functions needed for a program are in different files. In this chapter, we will see some of the issues in breaking up a program into functions, possibly spread over multiple files, but yet able to call each other and work together as a single program.

One important challenge is the management of names. A name is not useful if there is any ambiguity as to what exactly it refers to. On the other hand, if a program is written cooperatively, there is a chance that different programmers might use the same name to define a function that they wrote. Or it might be that you are borrowing a package (such as Simplecpp) written by someone else and using it in your program. Effectively, this also sets up the possibility of a name clash: how do you ensure that you don't use the same name that is used in the package that you borrowed? Or if you do use it, will it cause any ambiguity? These are some of the issues discussed in this chapter.

We begin by showing how Simplecpp turns the `main_program` that you write into a function as required by C++. After that we will see the rules for breaking up a program into multiple files, or alternatively, for assembling a program given a set of functions spread over several files. As mentioned above, an important issue will be management of names. The notion of *declarations* will be useful in this. Another useful notion will be that of a *namespace*, which we will also study.

We mentioned in the introduction that Simplecpp hides some of the technically advanced features of C++ so as to make it easier for a beginner to get to the heart of our subject: how to write programs.

[1] We will amend this statement a bit later

But this chapter will have introduced you to all the technical features hidden by Simplecpp. Thus by the end of the chapter, you will be able to use C++ directly, if you so choose, without having to include Simplecpp in your programs. We will discuss this in Section 11.6.

We will end by making some philosophical remarks on how to break a program into functions.

11.1 THE MAIN PROGRAM IS A FUNCTION!

C++, in fact, requires that the main program be written as the body of a function called `main`, which takes no arguments and returns an integer value. We did not tell you all this at the beginning of the book because at that time you did not know about functions, and it might have been too overwhelming to find out. So instead, we asked you to write the main program in a block following the name `main_program`. Our package Simplecpp uses the preprocessor facility (Appendix H) of C++ to change the name `main_program` that you write into the phrase `int main()`. Thus what you specify as the main program becomes the body of a function named `main` as required.

When a C++ program is compiled and run, the operating system expects that there be a function in it called `main`. *Running a program* really means calling the function `main`. The `main` function has return type `int` because of some historical reasons not worth understanding. You may also be wondering why we haven't been writing a `return` statement inside `main` if in fact it is supposed to return an `int`. The C++ compiler we have been using, the GNU C++ compiler, ignores this transgression, that's why!

Now that you know that the main program is just another function, from now on we will drop the name `main_program` and start writing the main program as a function named `main`. So should you.

11.2 ORGANIZING FUNCTIONS INTO FILES

It is possible to place the main program and the other functions in different files if we wish. If a program is very large, breaking it up into multiple files makes it easier to manage. A program can be partitioned into a collection of files provided the following rules are obeyed.

- **Rule 1** If a certain function `f` is being called by the code in file `F`, then the function `f` must be *declared* inside `F`, textually before the call to `f`. Note that a function definition is a declaration, but not vice versa. We will see what a declaration is shortly.
- **Rule 2** Every function that is called must be defined exactly once in some file in the collection.

Once you have a collection of files satisfying the above rules, they can be compiled into a program provided they contain a function `main`. We will see this with an example shortly.

11.2.1 Function Declaration or Prototype

The declaration of a function states the name of the function, its return type, and the types of its arguments. Indeed, a function declaration can be specified by giving its definition without the body. Here, for example, are the declarations of `lcm` and `gcd`.

```
int lcm(int m, int n);
int gcd(int m, int n);
```

The names of the parameters can be omitted from declarations, e.g. you may write just `int lcm(int,int);` in the declaration. The declaration is also called a *prototype*, or even a *signature*.

Suppose a compiler is processing a file containing the statement `cout << lcm(24,36);`. To process this correctly, the compiler needs to know that `lcm` is indeed a function, and not some typing mistake. It also needs to know the type of the value returned by `lcm` – depending upon the type the printing will happen differently. Both these needs are met by the declaration. A declaration of a function `f` provides (a) an assurance that `f` as used later in the program is indeed a function, and that in case it has not been defined so far, it will be defined later in this file itself or in some other file, (b) a description of the types of the parameters to the function and the type of the value returned by the function. Given the declaration, the file can be compiled except for the code for executing the function itself, which can be supplied later (Section 11.2.3). Notice that a function definition also provides the information in (a), (b) mentioned above, and hence is a also considered to be a declaration.

11.2.2 Splitting a Program into Multiple Files

Using the notion of a declaration, we can break up programs into multiple files. As an example, consider the main program of Section 9.2 that calls our function `lcm` to find the LCM of 36 and 24. Thus, there are 3 functions in our program overall: the function `main`, the function `lcm` and the function `gcd`. Figure 11.1 shows how we could form three files for the program.

First, consider the file `gcd.cpp`, which contains the function `gcd`. It does not call any other function, and so does not need to have any additional declaration. Next, consider the file `lcm.cpp`. This contains the function `lcm` which contains a call to `gcd`. So this file has a declaration of `gcd` at the very beginning. Finally, the file `main.cpp` contains the main program. This calls the function `lcm`, so it contains a declaration of `lcm` at the beginning. Note that the main program uses the identifier `cout` to write to the console. For this it needs to include `simplecpp`. The other files do not contain anything which needs services from Simplecpp, so those do not have the line `#include <simplecpp>` at the top.

There are various ways in which we can compile this program. The simplest possibility is to issue the command

```
s++ main.cpp lcm.cpp gcd.cpp
```

This will produce an executable file which will indeed find the LCM of 36,24 when run.

11.2.3 Separate Compilation and Object Modules

But there are other ways of compiling as well. We can separately compile each file. Since each file does not contain the complete program by itself, an executable file cannot be produced. What the compiler will produce is called an *object module*, and this can be produced by issuing the command

```
s++ -c filename
```

The option `-c` tells the compiler to produce an object module and not an executable. Here, `filename` should be the name of a file, say `main.cpp`. In this case, an object module of name `main.o` is produced. If different programmers are working on different files, they can compile their files separately giving the `-c` option, and they will at least know if there are compilation errors.

We can form the executable file from the object modules by issuing the command `s++` followed by the names of the object modules. Thus, for our example, we could write:

```
s++ main.o gcd.o lcm.o
```

```
//-------------------------------------------------------------------
int gcd(int m, int n){
  int mdash,ndash;

  while(m % n != 0){
    mdash = n;
    ndash = m % n;
    m = mdash;
    n = ndash;
  }
  return n;
}
//-------------------------------------------------------------------
```

(a) The file `gcd.cpp`

```
//-------------------------------------------------------------------
int gcd(int, int);          // declaration of function gcd.

int lcm(int m, int n){
  return m*n/gcd(m,n);
}
//-------------------------------------------------------------------
```

(b) The file `lcm.cpp`

```
//-------------------------------------------------------------------
#include <simplecpp>
int lcm(int m, int n);     // declaration of function lcm.

int main(){
  cout << lcm(36,24) << endl;
}
//-------------------------------------------------------------------
```

(c) The file `main.cpp`

Fig. 11.1 *The files in the program to find LCM*

This use of `s++` is said to *link* the object modules together. The linking process will check that every function that was declared in a module being linked is defined either in the same module or in one of the other modules being linked. After this check, the code in the different modules is stitched up to form the executable file.

It is acceptable to mix `.cpp` files and `.o` files as arguments to `s++`, e.g. we could have issued the command

```
s++ main.cpp gcd.o lcm.o
```

This would compile `main.cpp` and then link it with the other files. The result `main.o` of compiling will generally not be seen, because the compiler will delete it after it is used for producing the executable.

11.2.4 Header Files

Suppose programmers M, G, L respectively develop the functions `main, gcd, lcm`. Then G has to tell L how to declare the function `gcd` in the file `lcm.cpp`. The most natural way of conveying this information is to write it down in a so called *header file*. A header file has a suffix `.h`, or a suffix `.hpp` or no suffix at all, like our file `simplecpp`, which is also a header file. A simple strategy is to have a header file `F.h` for every program file `F.cpp` which contains functions used in other files. The file `F.h` merely contains the declarations of all the functions in `F.cpp` that are useful to other files. Thus, we might have files `gcd.h` containing just the line `int gcd(int,int)`, and `lcm.h` containing the line `int lcm(int,int)`. Now the programmer L writing the function `lcm` can read the file `gcd.h` and put that line into `lcm.cpp`. However, it is less error-prone and hence more customary that L will merely place the inclusion directive

```
#include "gcd.h"
```

in his file instead of the declaration. This directive causes the contents of the mentioned file, (`gcd.h` in this case) to be placed at the position where the inclusion directive appears. The mentioned file must be present in the current directory (or a path could be given). Thus all that is needed in addition is to place the file `gcd.h` also in the directory containing `main.cpp`. Likewise, M will place the line `#include "lcm.h"` in `main.cpp`, as a result of which the declaration for `lcm` would get inserted into the file `main.cpp` as needed.

Note that we could have used a single header file, say `gcdlcm.h`, containing both declarations.

```
int gcd(int,int);
int lcm(int,int);
```

We could include this single file in `main.cpp` and `lcm.cpp`. This will cause both declarations to be inserted into each file, while only one is needed. Having extra declarations is acceptable.

Note that the name of the file must typically be given in quotation marks if the file is present in the current directory or at a place given using an explicit path. If the file is present in some directory that the compiler is asked to look into using some other mechanisms, then angled braces are used, e.g. `#include <simplecpp>`. To get the exact details, you will need to consult the documentation of your compiler/linker.

11.2.5 Header Guards

Header files can become quite involved. If there are several header files, then you can place the inclusion directives themselves in another header file. Including the latter file will cause the former files to be included. If we have header files included inside one another, it raises the following possibility: we include some file `a.h` which in turn includes files `b.h` and `c.h`, both of which include the file `d.h`. This can cause a problem because as per our current definition of header files, whatever is in `d.h` will get included twice, and hence defined twice. Declaring the same name again is of course an error.

So what we need is a way to say, "include what follows only if it has not been included earlier".

This is what header guards provide. Indeed, it is more customary to write the header file `gcdlcm.h` in the following form.

```
#ifndef GCDLCM_H
#define GCDLCM_H
int gcd(int,int);
int lcm(int,int);
#endif
```

The first line checks if a so-called preprocessor (Appendix H) variable `GCDLCM_H` has already been defined. Only if it is *not* defined, then the rest file, till the line `#endif` is processed. Notice that the second line, should it be processed, will define `GCDLCM_H`. This will ensure that subsequent inclusions of the file `gcdlcm.h` will have no effect.

Note that preprocessor variables, unlike C++ variables, can be defined without being assigned a value, as in the code above. Also note that the name of the variable can be anything of your choosing; using the capitalized name of the file with a suffix `_H` is just a convention.

11.2.6 Packaging Software

The above discussion shows how you could develop functions and supply them to others. You place the functions in an `F.cpp` file and their declarations in an `F.h` file. Then you compile the `F.cpp` file giving the `-c` option. Then you supply the resulting `F.o` file and the `F.h` file to whoever wants to use your functions. They must place the file `F.h` in the directory containing their source files (i.e. files containing their C++ programs), and place the line `#include "F.h"` in the files which need the functions declared in `F.h`. Next, they must mention the file `F.o` while compiling, giving the pathname in case it is not in the current directory. Thus other programmers do not need to see your source file `F.cpp` if you don't wish to show it to them.

11.2.7 Forward Declaration

Let us go back to the case in which all functions are in a single file. We suggested in Section 9.1 that a function must be defined before its use (i.e. the call to it) in the file. However, as we discussed in the beginning of Section 11.2, it suffices to have a declaration before the use. So if we wish to put the function definition later, we must additionally place a declaration earlier.

When writing a program with several functions in a single file, many people like to place the main program first, perhaps because it gives a good overview of what the entire program is all about. We can do this; it is fine to organize the contents of your file in the following order.

```
declarations of functions in any order.
definition of main
definitions of other functions in any order.
```

Of course, other orders are also possible.

11.3 NAMESPACES

We next consider an important question which typically arises when a program makes use of functions developed by different people, developed possibly at different times. The question is: what happens if you borrow code from two programmers, both of whom have defined a function with the same name and the same signature? Note that you may not get the source code for the functions, and hence there may be no way of changing the name of any of the functions.

Such conflicts are typically resolved using the notion of a *namespace*. A namespace is like a catalog. When you place a name `name` in a namespace `catalog`, the actual name you define is not `name`, but `catalog::name`. Even after a name is placed in a namespace, it is usually possible to use just the short version `name` rather than always needing to use the full version `catalog::name`; we will see how this can be done in Section 11.3.2. However, the full name `catalog::name` is always available to use if the need arises. The operator `::` is called the scope resolution operator.

It is expected that if you create functions for public use, you will place them in a suitably named namespace. Thus, if you borrow code from two programmers, then quite likely they will use either different function names or different namespaces. Thus, even if the function names happen to be the same, by using the full name you can unambiguously refer to each function.

Next we see how to create and use namespaces. We consider only the main ideas and omit many details. How namespaces relate to header files is considered in Section 11.3.5.

11.3.1 Definition

You can define a namespace named `name-of-namespace` and the names inside it by writing

```
namespace name-of-namespace{
  declarations or definitions of names
}
```

Inside the block following the name `name-of-namespace` you can declare or define as many names as you like. For example, you might write

```
namespace mySpace{
  int gcd(int,int);
  int lcm(int m,int n){return m*n/gcd(m,n);}
}
```

After you write this, the namespace `myspace` will contain the names `gcd` and `lcm`. It is acceptable to give just a declaration (as we have done for `gcd`) or the complete definition (as we have done for `lcm`). Inside the namespace block, you can refer to the names in it directly. Thus, the definition of `lcm` refers to `gcd` directly. However, outside the namespace block, by default you must use the full name. For example, after the definition of `mySpace` above, you may define `gcd` by writing the following:

```
int mySpace::gcd(int m, int n){
  if(n == 0) return m;
  else return gcd(n, m % n);
}
```

You will note that the last line of the above definition uses the name `gcd` without prefixing it with the name of the namespace. This is fine; the definition of a function belonging to a namespace is considered to be an extension of that namespace, as a result other functions from that namespace can be referred to directly by their short names.

You can also give the definition by writing the following:

```
namespace mySpace{
  int gcd(int m, int n){
    if(n == 0) return m;
```

```
    else return gcd(n, m % n);
  }
}
```

Indeed you can add to a namespace as many times you wish by writing more namespace blocks with the same name.

Finally, we can use the functions in the namespace by writing out their full names.

```
int main(){
  cout << mySpace::lcm(36,24) << endl;
}
```

This must follow the namespace definition and the declaration of `mySpace::lcm`.

11.3.2 The `using` Declaration and Directive

It becomes tedious to keep writing the full name of a function. However, we can use the shorter form by including a so-called *using* declaration:

```
using ns::n;
```

Here, `ns` is the name of a namespace, and `n` a name defined inside it. In the code following the using declaration, all references to the name `n` would be considered to be referring to `ns::n`. Thus, we might write

```
using mySpace::lcm;
```

If we put this line before the main program, then in the main program, we can use `lcm` rather than having to write `mySpace::lcm`.

Another variant is to merely state

```
using namespace ns;
```

This is called a namespace directive. With this, all names in the namespace `ns` can be used using the short form in the code that follows.

11.3.3 The Global Namespace

When you define functions without putting them in a namespace, they implicitly enter an omnipresent, nameless *global namespace*. When you use a name without a namespace qualifier, it is expected to be present in either the global namespace or in a namespace for which an appropriate using declaration or directive has been given.

Suppose we have the namespace `mySpace` as defined above, and further we have put a `using namespace mySpace;` directive alongwith our main program, as follows.

```
using namespace mySpace;

int lcm(int m,int n){return m*n;}  // not really computing the lcm!

int main(){
  cout << lcm(36,24) << endl;
}
```

In this case, the compiler will flag an error, saying that the reference `lcm` in the main program is ambiguous, it could refer to the `lcm` function in the global namespace, or the lcm function in `mySpace` which has been made available through the using directive. In such a case, you must give the full name of the function and in doing so, pick one.

To specifically refer to a function `lcm` in the global namespace, you can write `::lcm`.

Thus in the main program above, you must change `lcm` to either `::lcm` or `mySpace::lcm`. Note that if the two `lcm` definitions had a different signature, then this problem would not have arisen (Section 12.4).

11.3.4 Unnamed Namespaces

We discuss a somewhat technical point which could be ignored.

If you define a function `f` in the global namespace in one file `F1.cpp`, it is potentially available for use in other files, say `F2.cpp`, provided you compile the files together, i.e. by issuing the command

```
s++ F1.cpp F2.cpp
```

while compiling.

Sometimes you might intend the function `f` to be used only within file `F1.cpp` and not be exposed outside. For this, you can define it inside an unnamed namespace by writing

```
namespace{            // notice that no name is given
  declaration of f
}
```

With this, you can use `f` inside the file in which the declaration appears, direclty by its name, but not outside of the file. You can think of an explicitly created nameless namespace as a global namespace available only to functions in the file in which the declaration appears.

11.3.5 Namespaces and Header Files

Generally, namespace definitions are made inside header files, and in such definitions only declarations are put. The function definitions are put in implementation files, in which the header file containing the namespace definition must be included.

The files that use the functions in the namespace must also include the file containing the namespace definition, and may use the functions in the namespace either by giving the full name or by giving a `using` directive and the short name.

Figure 11.2 shows an example.

11.4 GLOBAL VARIABLES

So far in this book we have had variables defined inside functions. However, it is possible, though not recommended, to define a variable outside all functions. In such cases, the variable becomes a *global* variable, i.e. it can be accessed by any function. Note that the compiler will typically have a separate region of memory where global variables will be allocated; global variables are *not* allocated in the activation frame of any function.

```
namespace mySpace{
  int gcd(int,int);
  int lcm(int,int);
}
```

(a) The file `mySpace.h`

```
#include "mySpace.h"

int mySpace::lcm(int m,int n){
  return m*n/gcd(m,n);
}
int mySpace::gcd(int m, int n){
  if(n == 0) return m;
  else return gcd(n, m % n);
}
```

(b) The file `impl.cpp`

```
#include <simplecpp>
#include "mySpace.h"

using namespace mySpace;
int main(){
  cout << lcm(36,24) << endl;
}
```

(c) The file `main.cpp`

Fig. 11.2 *Program in multiple files using namespaces*

Here is an example.

```
int i=5;                    // global variable definition

void f(){ i = i * i; }      // refers to global variable

int main(){
  cout << i << endl;        // refers to global variable
  f();
  cout << i << endl;        // refers to global variable
}
```

If you execute this code, the first statement will print 5, the current value of the global variable `i`. Then the call `f()` inside the main program with change `i` to 25, which will be printed by the third statement.

Use of global variables is not encouraged, because global variables make code hard to understand. Potentially, any function call could modify the variable, and hence, it is difficult to make claims about the value taken by the variable at any point of the execution. However, global variables are used in several (especially older) programs, and hence, this discussion.

We note that a global variable defined in one file can be used in another file as well. However, it must be declared as `extern` in that file. Thus, to use the global variable `i` defined above in another file, that file would need to have the declaration

```
extern int i;                    // declaration, not definition
```

Note that this does not define space for `i`, it merely declares `i` to be of type `int` which will be defined in some other file.

You may also put global variables in namespaces. The simplest way is to place the declaration (which is merely the definition prefixed by `extern`) inside the namespace block, which can be in a header file. Then one of the implementation files should contain the definition of the variable.

We finally note that individual functions may contain definitions of a local variable having the same name as a global variable. In such cases, the local variable will *shadow* the global variable (Section 3.6.3).

11.5 TWO IMPORTANT NAMESPACES

The most important namespace that C++ programmers need to know about is the namespace `std`. As you might guess `std` stands for "standard".

The namespace `std` defines many names that you might have so far been considering to be reserved words. Indeed, the words `cin`, `cout`, `endl`, as well as the words `string` and others that you will see soon are in this namespace. But if `cin` is in `std`, you may wonder why you have not been forced to write `std::cin` instead of just `cin` so far. The answer, as you might guess, is that the file `simplecpp` that you include when you write

```
#include <simplecpp>
```

contains the directive `using namespace std;`.

Another namespace important for this book (but not for C++ in general) is the namespace `simplecpp`. All names such as `initCanvas`, `Circle`, `Line`, `forward`, `left` are in this namespace. As you may guess, you can use these names directly because the file `simplecpp` contains the directive `using namespace simplecpp;` also.

11.6 C++ WITHOUT Simplecpp

We now consider the question of how C++ programs can be written and compiled without using `simplecpp`. We show an example file in Figure 11.3.

This file, `area.cpp` can be compiled using any C++ compiler, say the GCC compiler. Most commonly the GCC compiler for C++ is invoked by the command `g++` as follows.

```
g++ area.cpp
```

Everything in the file should look familiar except for the first two lines. The file `iostream` is a header file that contains declarations of names needed for performing input output. The file `cmath` is a header file that contains declarations of mathematical functions such as `sqrt` and also other functions including trigonometric functions. You may wonder why did we not need to include these files so far – and the answer of course is that we included the file `simplecpp` which was in turn including these two files.

```
#include <iostream>
#include <cmath>
using namespace std;

int main(){
  cout << "Give area of square: ";
  double area;
  cin >> area;
  cout << "Sidelength is: " << sqrt(area) << endl;
}
```

Fig. 11.3 *Contents of the file* `area.cpp`

If you program does not contain graphics, then you can dispense with `simplecpp` if you wish, as seen above for the file `area.cpp`. All you need to do is that instead of including `simplecpp`, you include the files `iostream` and `cmath`, and also put in the `using` directive. Also you also cannot use the command `repeat`; but we have already suggested that you start using the other looping commands (Chapter 7) instead. Finally, you should not define the main program as `main_program` but defines it as a function named `main`. There are a few other minor features in `simplecpp` that you cannot use – and we will discuss these as we encounter them.

11.7 THE `exit` FUNCTION

Normally, the program terminates when the control reaches the end of the `main` function. However, you may wish to end execution prematurely, say because you have discovered some serious problem. You can do so by calling the `exit` function. For example, you may write

```
if(numberOfvariables < 0) exit(1);
```

The function `exit` takes an integer argument for historical reasons, which we have supplied above as 1. To use this function you must include the header file `<cstdlib>`. If you are including `simplecpp`, you need not worry, `<cstdlib>` will be included as a part of `simplecpp`.

The function `exit` can be called from anywhere in your program, not just from the `main` function.

11.8 CONCLUDING REMARKS

This chapter dwelled on many technical aspects of using functions. These are important especially while writing large programs. It is inevitable that large programs will be written in multiple files. We saw how *header files* can be used to make functions in one file visible to code in other files. For ease of writing header files, it is convenient to use header guards, which you must understand and master.

Another idea that helps in building large programs is that of namespaces. We saw how to construct namespaces, and how they can be used to prevent conflicts even if several people cooperating on a program happen to use the same name.

We also saw how C++ can be used without Simplecpp.

We have been discussing functions as a mechanism for avoiding code duplication and for splitting work amongst multiple programmers. However, there is another important motivation for writing functions; by splitting your code into several small functions, it may be possible to make it easier to understand. This is an extremely important motivation, and we remark on it below.

11.8.1 Function Size and Readability

One way to improve understandability of anything is to present it in small chunks. When you write a book it is useful to break it up into chapters. A chapter forms an organizational unit of a book. In a similar manner, a function forms an organizational unit of a program. A function can become more understandable if its code can be broken up into smaller functions, each with a name describing what that piece does. There are a few thumb rules for breaking long text into chapters or sections. An example of a thumb rule: every idea that is important should have its own chapter, or its own section. There are similar thumb rules for splitting large programs into functions.

Many programmers believe that no function, including `main` should be longer than one screenful. Even with large displays, this gives us a limit of perhaps 40–50 lines on the length of a function. Basically, you should be able to see the entire logic of the function at a glance: that way it is easier to understand what depends upon what, or spot mistakes. How do we break a program into smaller pieces? So far you have not had the occasion to write programs longer than 40–50 lines, so this discussion is perhaps a bit difficult to appreciate. You will see later, however, that most programs can be thought of as working in phases. Then you should consider writing each phase as a separate function, and give it a properly chosen name that describes what it does. These functions could be placed in the same file as the main program. You will find that this will make the program easier to understand. Another idea is to make a function out of any modestly complicated operation you may need to perform. As an example of this, consider the apparently simple action of reading in a value from the keyboard. A user might type in an invalid value, or the value may not stand for itself but in fact might indicate that the stream of values has finished. These are details that are unimportant for understanding the rest of the program. So you could hide away the logic for dealing with all this in a function that is called by the main program. This idea is partly explored in the `read_marks_into` function of Section 9.7 and `readFromRange` function of Section 9.4.1.

EXERCISES

1. Suppose the LCM computation program of Figure 11.1 has been written using a single file, and it is noticed that only the function `lcm` has been declared and also defined, all other functions are defined but not declared. Show how the program could appear in the file.
2. Suppose you have a computer on which you cannot divide for some reason. You wish to determine if a number is even. You could base your program on the following observations.
   - The number 0 is even.
   - A number x is even if $x - 1$ is odd.
   - A number x is odd if $x - 1$ is even.

   Write mutually recursive functions `even` and `odd` as suggested above. You will need to declare the functions appropriately.
3. Organize the code of the previous exercise in multiple files as follows. There should be one header file called `functions.h`, and files `main.cpp`, `even.cpp` and `odd.cpp` which contain the main program, the function `even` and the funcion `odd` respectively. Each of the `.cpp` files should include `functions.h`. Use header guards as appropriate inside `functions.h`.
4. Define a namespace `mySpace` and put the `even`, `odd` functions in it. Write a main program which calls `even` to determine if a given number that the user has typed is even. Do this using the `using` directive, the `using` declarations, and without using `using` at all.

CHAPTER 12

Functions: Advanced Topics

In this chapter, we will consider some advanced topics related to functions. You can write programs without knowing what we discuss in this chapter. However, as you will see, the constructs we discuss in this chapter may often help you *directly* express what you want, and this will make for more succinct and understandable programs. Succinctness and understandability are both important and worth striving for.

It often turns out to be useful to write a function which takes another function as an argument. Why this is needed, and how it can be done, is discussed in Section 12.1. We also discuss the notion of *lambda expressions*, which allow you to define and manipulate functions without giving them names. Section 12.1 gives a motivating example for this also. Lambda expressions will make for succint expression in many programs, and you will see examples of this in Section 22.3.1, Section 22.3.2, Chapter 27 and Chapter 28.

We next consider some conveniences provided in defining functions. For example, with certain restrictions, it is possible to give the same name to two functions. Also, it is possible to define functions in which some arguments are given some default values if they are not specified in the call. We discuss these features and the circumstances in which they are useful in Section 12.3 and Section 12.4.

The last topic in the chapter is that of function *templates*. Sometimes you realize that at an abstract level, the same algorithm can be used for arguments of different types. In such cases, you can define a function template, from which functions for different types can be generated as and when needed. This is discussed in Section 12.5.

12.1 FUNCTIONS AS ARGUMENTS

In Section 8.3, we discussed the bisection method for finding the roots of an equation, or equivalently, a zero of a mathematical function, i.e. a value of x such that $f(x) = 0$. The program we wrote was for finding the root of the equation $x^2 - 2 = 0$, or alternatively, the zero of $f(x) = x^2 - 2$. But the method could be used for finding zeroes of other functions too, e.g. of $g(x) = \sin x - 0.3$. The code for finding a zero of f and of g would be the same, except in the place where we evaluate the function itself. This raises the question: could we write the bisection method as a C++ function `bisection`, to which the mathematical function whose zero is to be found will be supplied as an argument?

Of course, to pass f, we must have a suitable representation for it. The simplest way is as a C++ function, say `f` which takes x as an argument and returns the value of $f(x)$. But then, we need a way of passing a C++ function, say `f`, as an argument to another C++ function, say `bisection`.

```
#include <functional>

double f(double x){
  return x*x -2;
}

double g(double x){
  return sin(x) - 0.3;
}

double bisection(double xL, double xR, double epsilon,
                 std::function<double(double)> pf)
// precondition: f(xL),f(xR) have different signs. ( >0 and <=0).
{
  bool xL_is_positive = pf(xL) > 0;
  // Invariant: x_L_is_positive gives the sign of f(x_L).
  // Invariant: f(xL),f(xR) have different signs.

  while(xR-xL >= epsilon){
    double xM = (xL+xR)/2;
    bool xM_is_positive = pf(xM) > 0;
    if(xL_is_positive == xM_is_positive)
      xL = xM;  // maintains both invariants
    else
      xR = xM;  // maintains both invariants
  }
  return xL;
}

int main(){
  double y = bisection(1, 2, 0.0001, f);  // Alternate syntax: &f
  cout << "Sqrt(2): " << y << " check square: " << y*y << endl;

  double z = bisection(0, PI/2, 0.0001, g);
    cout << "Sin inverse of 0.3: " << z << " check sin: "
    << sin(z) << endl;
}
```

Fig. 12.1 *Bisection method as a function*

Figure 12.1 shows how this can be done. This code contains C++ equivalents of two mathematical functions, $f(x) = x^2 - 2$, and $g(x) = \sin x - 0.3$. The single function `bisection` is used to find the zeroes of both these functions. We will explain how `bisection` works shortly, but first consider

the main program. As you can see, `main` calls `bisection` for finding each root. Consider the first call. The first two arguments to the call are the left and right endpoints of the interval in which we know the function changes sign. As in Section 8.3, we have used the values 0,2 for the left and right endpoints. The next argument gives the acceptable error in the function value. For this we have chosen the value 0.0001, instead of reading it from the keyboard as in Section 8.3. The last argument is the function `f` whose root is to be computed. The second call is similar. We are asking to find a root in the interval 0 to $\pi/2$, where we know that $g(0) < 0$ while $g(\pi/2) > 0$, as required for the bisection method. The error tolerance is 0.0001. The last two lines merely print out the answers and check if the square of the first answer is close to 2, and the sine of the second answer is close to 0.3

We now discuss the function `bisection`. Its first three parameters `xL, xR, epsilon` are self explanatory. The fourth parameter is meant to be a function. Its type declaration is a bit tricky. This can be done in two ways. We describe the modern, easier method first. But for this you need to include the header `<functional>` as is done at the top of Figure 12.1. Using the features from `<functional>`, you can specify a function type by writing

```
std::function<returntype(param1type, param2type, ...)>
```

Here, `returntype` must be the type of the value returned by the function, and `param1type, ...` must be the types of the parameters of the function. In the present case, we expect `bisection` to be passed a function which takes a `double` as an argument and returns a `double`. Hence, we have specified this as `std::function<double(double)>`. This is followed by the name of the parameter, `pf` in this case. Now within the body of `bisection` we can make calls to `pf` assuming it takes a `double` argument and returns a `double`.

12.1.1 Function Pointers

You might wonder what it *means* to pass a function as an argument. As you know from Section 2.9, code and data are both placed in memory. Just as we can identify a variable by its starting address, we can identify a function also by the address from where its code starts. Indeed, C++ has the notion of a *function pointer*, which you could think of as the starting address of the code corresponding to the function. The *original* syntax for passing functions in fact requires you to write a function argument as `&f`, meaning the address of a function `f` rather than as `f`. But modern compilers are permissive and allow you to drop the `&` while calling, as we have done in the main program of Figure 12.1. Further, the original syntax requires that you first dereference the function parameter before using it; thus you would write `*pf` when using the function parameter in a call in `bisection`. But this is also not necessary, you are allowed to drop the derererencing operator. Note by the way that the original syntax is still acceptable, and you may see code explicitly using the `&` and `*` operators.

The original syntax is also different as far as declaring a function parameter. To understand this, let us first review function declarations. Here is a declaration of a function `pf` that takes a `double` as argument and returns a `double` result (Section 11.2.1).

```
double pf(double);  // pf is a function taking double
                    // and returning double
```

Next we simply note the general strategy for declaring pointers: if a declaration declares name `v` to be of type `T`, then replace `v` by `*v` and the new declaration will declare `v` to be pointer to T. Doing this, we get what we wanted.

```
double (*pf)(double);  // *pf is function taking double and
                       //  returning double
 // pf is pointer to function taking double and returning double
```

where the parentheses have been put because we want * to associate first with pf.[1] Thus we would write bisection as

```
double bisection(double xL, double xR, double epsilon,
                 double (*pf)(double x))
{
...
}
```

This syntax is still acceptable, and you may use it. Of course, you may find the syntax std::function<...> easier to understand. If so, go ahead and use just that. But be sure to include <functional>.

12.2 NAMELESS FUNCTIONS: LAMBDA EXPRESSIONS

The program of Figure 12.1 can be written more naturally and compactly by using so called *lambda expressions*, which we discuss next.

A lambda expression is a nameless function which can be constructed pretty much anywhere in your code and subsequently used. The following expression, for example, represents a function which works like the function f of Figure 12.1, it can be used to compute $x^2 - 2$:

```
[](double x){return x*x - 2;}
```

We will explain the syntax shortly. But we first note that lambda expressions can be used wherever a function name can be used, e.g. we can call it by passing arguments to it:

```
[](double x){return x*x - 2;}(3.5)
```

This will evaluate to $3.5^2 - 2 = 10.25$. Alternatively, we can even place it in a call to bisection of Figure 12.1:

```
  double y = bisection(1,2,0.0001, [](double x){return x*x -2;});
```

Thus, we would not need to define the function f at the beginning of Figure 12.1, but place its code directly where it is needed inside main. Likewise the function g.

The C++ standard adopted in 2011 supports lambda expressions. These have many uses, e.g. see Section 22.3.1, Section 22.3.2, Chapter 27 and Chapter 28. Lambda expressions were originally proposed in languages such as LISP, where they got the name.

12.2.1 General Form

Lambda expressions can be specified in many ways. We first discuss the form used above.

```
[](parameter-list){body}
```

[1] If we omit the parentheses, we would be declaring pf to be a function which takes a double argument and returns a pointer to a double.

In this, `parameter-list` gives the list of parameters that the function needs, and the `body` gives the code that is to be executed. You have already seen an example. Something can be specified inside the leading `[]`, we will see this shortly.

Note that we have not explicitly stated the return type of the function. C++ will infer this on the basis of the `return` statements in `body`. Sometimes, C++ may not be able to infer correctly, in which case you can specify the return type using the following syntax.

```
[](parameter-list) -> return-type {body}
```

Thus, you could write

```
[]() -> int {return 1;}
```

Since 1 can have many types, the above clarifies that we mean a function which returns an `int`.

12.2.2 The Type of a Lambda Expression

A lambda expression can be considered to have the type

```
std::function<return-type(parameter-types)>
```

where `return-type` is the return type of the lambda expression and `parameter-types` are the types of the parameters (comma separated). To use this, you must include the header file `<functional>`.

12.2.3 Variable Capture

The body of a lambda expression may access names which are defined outside the lambda expression, but are visible in the scope in which the lambda expression is defined. Such names are sometimes called *free* names of the lambda expression (as opposed to those variable names that are defined or bound inside the lambda expression). The free names can either denote the values of the corresponding variables at the time the lambda expression was defined, or denote references to the corresponding variables. The former is called "capture of free names by value" and the latter "capture of free names by reference". To denote capture by value, we simply give the names in the initial `[]`, to denote capture by reference, we give the names prefixed by `&`. Here is an example.

```
int main(){
  int m=10;
  std::function<void()> f = [m](){cout << m << endl;};
  std::function<void()> g = [&m](){cout << m << endl;};

  m++;

  f();
  g();
}
```

In this, `f,g` capture `m` by value and by reference respectively. Thus the call `f()` will print 10, which is the value of `m` at the time `f` got defined. The call `g()` on the other hand will print 11, which is the current value of `m` since `g` has captured `m` by reference.

If you wish to capture `a` by reference and `b` by value, you may specify the capture as `[&a,b]`. If you want all to be captured by value or all by reference you may specify the capture as `[=]` or `[&]` respectively. If you want all to be captured by value except for `a,b`, you may write `[=,&a,&b]`, and analogously. Note that writing the capture as `[]` specifies no capture.

We will see intersting examples of variable capture in Chapter 27.

12.2.4 Dangling References

If you capture a variable by reference, and the variable is deallocated between the capture and the use, then we have the problem of a dangling reference like Section 9.9.1. Thus capture by reference must be done carefully.

12.3 DEFAULT VALUES OF PARAMETERS

It is possible to assign default values to the parameters of a function. If a particular parameter has a default value, then the corresponding argument may be omitted while calling it. The default value is specified by writing it as an assignment to the parameter in the parameter list of the function definition. Here is a `drawPolygon` function in which both parameters have default values.

```
void drawPolygon(int nsides=4, double sidelength=100)
{
  for(int i=0; i<nsides; i++){
    forward(sidelength);
    right(360.0/nsides);
  }
  return;
}
```

Given this definition, we are allowed to call the function either by omitting the last argument, in which case the `sidelength` parameter will have value 100, or by omitting both parameters, in which case the `nsides` parameter will have value 4 and `sidelength` will have value 100. In other words, we can make a call `drawPolygon(5)` which will cause a pentagon to be drawn with side length 100. We can also make a call `drawPolygon()` for which a square of sidelength 100 will be drawn. We are free to supply both arguments as before, so we may call `drawPolygon(8,120)` which will cause an octagon of sidelength 120 to be drawn.

In general, we can assign default values to any suffix of the parameter list, i.e. if we wish to assign a default to the ith parameter, then a default must also be assigned to the $i+1$th, $i+2$th and so on.

Further, while calling we must supply values for all the parameters which do not have a default value, and to a prefix of the parameters which do have default values. In other words, if the first k parameters of a function do not have default values and the rest do, then any call must supply values for the first j parameters, where $j \geq k$.

Note also that the default values must also be specified in the function prototype.

12.4 FUNCTION OVERLOADING

C++ allows you to define multiple functions with the same name, provided the functions have different parameter type lists. This comes in handy when you wish to have similar functionality for data of

multiple types. For example, you might want a function which calculates the `gcd` of not just 2 numbers, but several, say 3 as well as 4. Here is how you could define functions for doing both, in addition to the `gcd` function we defined earlier.

```
int gcd(int p, int q, int r){
  return gcd(gcd(p,q),r);
}

int gcd(int p, int q, int r, int s){
  return gcd(gcd(p,q),gcd(r,s));
}
```

The above functions in fact assume that the previous `gcd` function exists. Here is another use. You migth want to have an absolute value functions for `double` data as well as `int`. C++ allows you to give the name `Abs` to both functions.

```
int Abs(int x){
  if (x>0) return x;
  else return -x;
}

double Abs(double x){
  if (x>0) return x;
  else return -x;
}
```

While it is convenient to have the same name in both cases, you may wonder how does the compiler know which function is to be used for your call. The answer is simple: if your call is `abs(y)` where `y` is `int` then the first function is used, if the type is `double` then the second function is used. Likewise, the right `gcd` function will be picked depending upon how many arguments you supplied.

12.5 FUNCTION TEMPLATES

You might look at the two `abs` functions we defined in the preceding section and wonder: sure the functions work on different types, but the bodies are really identical, could we not just give the body once and then have the compiler make copies for the different types? It turns out that this can be done using the notion of *function templates* as follows.

A function template does not define a single function, but it defines a template, or a scheme, for defining functions. The template can have parameters: by specifying suitable values for the parameters, a function will get constructed! Here is an example.

```
template<typename T>
T Abs(T x){
  if (x>0) return x;
  else return -x;
}
```

The first line of this states that what follows is a template in which the name `T` is the template parameter. The parameter name is prefixed with `typename`, which says that `T` must be given a value which must be the name of a type, say `int`. As you can see, you will indeed get a function if you substitute `int` for `T` in the rest of the template. The keyword `class` can also be used instead of `typename`.

C++ will automatically construct a functions from this template, depending upon how you call `Abs` in your code. If you call `Abs` with an `int` value, then C++ will construct an `Abs` function for you by substituting `int` for `T` in the template. If you have another call which calls `Abs` with a float `argument`, then C++ will generate another `Abs` function, this time by substituting `float` for `T`. Here is an example.

```
int main(){
  int x=3;
  float y=-4.6;

  cout << Abs(x) << endl;
  cout << Abs(y) << endl;
}
```

This main program will work along with the template given above. C++ will construct the required functions without you having to do anything.

You can have templates with several parameters. Values will be substituted for them depending upon the call. All template parameters need not have type `typename`. You may have other types too, e.g. `int`. An example of this is given in Section 18.11.5.

12.5.1 Function Templates and Header Files

A function template must be present in every source file in which a function needs to be created from the template. So a template is best written in a header file. Note that the template itself cannot be compiled; only the function generated from the template is compiled. So for a template function `f` we will typically have a header file `f.h`, but no file `f.cpp`.

12.6 CONCLUDING REMARKS

When we introduced functions in Chapter 9, you may have thought of functions as being very different from variables. Functions are different, however, over the last few chapters, you can perhaps see a unifying trend. First, we associated types with functions. In this chapter we allowed functions to be passed around just as we pass around variables. Finally, we also defined the notion of a nameless function, this can be considered similar to the notion of a literal, a value without a name. This trend of unifying functions and variables is not a merely technical exercise. As we saw, functions which can take other functions as arguments and lambda expressions allow us to say what we want very simply and directly. So you must master these ideas. Incidentally, a matter of terminology: functions which take other functions as arguments are sometimes called *higher order functions*.

We also saw other conveniences such as the ability to give the same name to two functions. Of course, it is important to use this judiciously. The same name should be given only if the functions abstractly serve the same purpose, but with different types. We also saw how to specify default values to arguments. This is also often very convenient.

Function templates have evolved out of the realization that often we write code the text of which can work for values of many types. Using templates, we can actually write the text once, and specify that it can work for any type. When the need arises, a function of that type will get created.

We remarked in Section 1.8 that an important activity in programming is observing and exploiting patterns. As you can see, higher order functions and templates take this idea one step further. In both cases there is the observation that the same code pattern is useful in many situations. Thus a facility is provided by which the code pattern can we expressed once and used as needed.

EXERCISES

1. Write a function to find roots of a function `f` using Newton's method. It should take as arguments pointers to `f` and also to the derivative of `f`. Test it by supplying it appropriate lambda expressions, and also names of functions that you have defined.
2. Write a function that plots the graph of a mathematical function on the graphics canvas. The mathematical function whose graph is to be plotted should be accepted as a function pointer or a lambda expression. Also the domain and range for the plot should be specified. You could also accept information about whether to put ticks on the axes etc. as arguments to the call. Test it by plotting various functions. Define default values for some of these parameters.
3. The k-norm of a vector (x, y, z) is defined as $\sqrt[k]{x^k + y^k + z^k}$. Note that the 2-norm is in fact the Euclidean length. Indeed, the most commonly used norm happens to be the 2 norm. Write a function to calculate the norm such that it can take k as well as the vector components as arguments. You should also allow the call to omit k, in which case the 2 norm should be returned.
4. Modify the letter-drawing function of Section 9.5 to take optional arguments giving the coordinates of the point from which the drawing should start. By default, the drawing should start from the current position of the turtle. You may wish to have the default value be -1 or any invalid value, in order to check whether the user has supplied a value.
5. The function passed to the `bisection` function took a `float` and returned a `float`. However, we might well need to find the root of a function which takes a `double` and returns a `double`. Also, it would be nice if the types of the other arguments could likewise be chosen to be `double`. Turn `bisection` into a template function so that it works for both `double` and `float` types. You can of course also do this by overloading the name `bisection`.

CHAPTER 13

Practice of Programming: Some Tips and Tools

In theory, theory and practice are the same. In practice, they are not.

ALBERT EINSTEIN

If you have been reading this book without solving any of the exercises, you may perhaps come to believe that the process of developing a program is neat and tidy, and generally smooth sailing. If on the other hand, you have also been solving the exercises, your experience might be different. After you write a program, you compile and run the program and you test it by providing different inputs. You possibly discover that for some input, let us call it x, the program does not produce the correct output. After some amount of detective work, you figure out the reason. So you modify your program (often called "debugging") and rerun. You then discover that now the program works correctly for x, but it does not run correctly for an input y, for which the old program was correct! So you have to do more detective work. And this cycle can go on for some time. If you are writing large programs, this cycle can be extremely frustrating.

In this chapter, we will discuss the entire programming process and offer suggestions with a view to (a) increase the confidence that the program is doing what it is supposed to do, and (b) make the process less tiresome.

The program development process starts with clearly understanding what is to be done, i.e. the specification for the program. Along with understanding the specification abstractly, it is useful to construct examples of inputs and required outputs. These help in ensuring that the specification is correct, and can also be used to test the program when it is written. After this comes program design, and then inevitably, debugging. We will discuss all these phases. Along the way, we will also remark on some general facilities like input-output redirection and assertions that are available to simplify the above phases. In some ways, this chapter is an extended version of Chapter 4.

Some of the suggestions made in this chapter may strike you as being too cautious. You may think, "I can go much faster than this". In case you are one of the lucky few who can write large programs correctly, in an intuitive manner, without apparent careful planning, more power to you! But if you find you are spending more time debugging the program than designing it in the first place, you are encouraged to try out the suggestions given in this chapter.

13.1 CLARITY OF SPECIFICATION

The first step in writing a correct program is to clearly know what you want the program to do. This might sound obvious, but often, programs don't work because the programmer did not clearly consider what needs to happen for some tricky input instance. How can you be sure that you completely *know* what the program is expected to do, that you have considered all the possibilities? The best way of doing this is to *write* down the specifications very clearly, in precise mathematical terms if possible.

Typically, in a specification you will state that the inputs consist of numbers $x, y, z, \ldots$, and the output consist of the numbers $p, q, r, \ldots$. Then you will give conditions that these numbers must satisfy. The specification is not expected to indicate how the output is to be actually generated, that is to be decided by your program, sometimes referred to as the *implementation*. If the output produced by your implementation happens to satisfy the conditions described in the specification for every input, then and only then can your implementation be certified as correct. It is a good idea to write the specification as a comment in your program, and also to use the same variable names in the specification as in your program.

Let us take an example. What are the specifications for the program to count digits of a number? We think that we understand decimal numbers, which we indeed do. But such intuitive understanding does not constitute a specification. The intuitive understanding must be stated in more precise terms. Here is the specification we used earlier.

Input: A non-negative integer n.

Output: Smallest positive integer d such that $10^d > n$.

This is a good specification because it gives the precise conditions that we want the output to satisfy, nothing more, nothing less.

It is customary in writing specifications to state conditions in the form "*smallest/largest... satisfying...* ". Formulating specifications in this manner requires some practice. Also a lot of care is needed. Should the condition be $10^d > n$ or $10^d \geq n$? You may consider these questions to be tedious, but if you cannot answer them correctly while writing the specification, you are unlikely to write the program correctly. You may be making the same mistake in your program!

Let us consider another problem. Suppose you are given n points in the plane, $p_1, \ldots, p_n$. Find the smallest circle that contains all the points. It might be tempting to rewrite just what is stated in the problem statement:

Input: n points $p_1, \ldots, p_n$ in the plane.

Output: Smallest circle that contains all points.

In this we have not specified how a circle is to be represented using numbers. That is acceptable, if our audience knows how to represent circles using numbers and translate the phrases such as "smallest circle", and "contains all points", into conditions on numbers. For the present, however, we will prefer the following description of the output.

Output: Real numbers x, y, R such that the distance between each point p_i and the point (x, y) is at most R, and R is smallest possible.

In this, we have not defined what "distance" means. If that is not expected to be commonly understood, then we should spell it out too.

Consider another example.

Input: n points $p_1, \ldots, p_n$ in the plane specifying the vertices of a polygon in clockwise or counterclockwise order.

Output: Area of the polygon.

This seems like a good specification. Although we have not given a formula to compute the area, the notion of area is common knowledge. Or is it? A good step after writing a specification is to consider whether it admits any "unexpected inputs". If you think about this, you may realize that the user may give a polygon whose edges intersect with each other, i.e. the polygon is not simple. If we allow non-simple polygons as input, the problem statement needs to define what area means for non-simple polygons. If the user expects to supply only simple polygons as input, the specification must state this.

Input: n points $p_1, \ldots, p_n$ in the plane specifying the vertices of a **simple** polygon in clockwise or counterclockwise order.

Even with this specification, it is possible that the user actually supplies a non-simple polygon as input. What should the implementation do then? One answer is nothing: this case is not a part of the specification, and so nothing is promised. So it could be argued that the implementer cannot be blamed if the program produces some strange value, or even if it goes into an infinite loop! The other possibility is that the implementer checks the given input and prints an error message stating that the input does not confirm to the specification, and then stops. Of course such checking might require more effort and time. But it may often be worthwhile.

The points to note are as follows. First, write down specifications as precisely as possible, using mathematical notation if it is reasonably obvious. Second, you may receive a specification that looks fine, but actually admits some unexpected inputs. In this case, you should modify the specification. Finally, the implementer must mindfully decide what to do if inputs that do not adhere to the specification are supplied during execution: whether to point this out or not take responsibility.

13.2 INPUT-OUTPUT EXAMPLES AND TESTING

Along with writing the specifications, you should construct sample input instances, and work out what output you want for those. As discussed in Section 4.1.1, it is good to have examples in your mind for any abstract statements you make. Another reason is that the input-output examples you work out will serve later as test cases for your program.

For the digit-counting program, it is easy to work out examples. For example, you might arbitrarily decide to have your first input instance be the number 34, for which the output must be 2 since that is the number of digits in 34. This might appear too easy, but even so it should be written down. You should also check whether the input (34) and the output (2) agree with the what you have written down in the specification: Is 2 indeed the smallest number such that $10^2 > 34$? These may sound like trivial checks, but your program can go wrong because of trivial mistakes, and so such checks are useful. For the circle covering problem, working out examples is harder, since it might take considerable calculation to find the smallest covering circle by hand. In such cases, the least you can do is to construct a few simple cases, e.g. just two points, say (0,0) and (1,0), for which the best covering circle must have radius 0.5 and must be centered at (0.5,1). You should even try "degenerate" cases like a single point input: the output should then be a circle of radius 0.

A more systematic approach is to try to figure out input instances which might occur "commonly". For example, for the covering circle problem, the instance in which all points are randomly placed in

the plane is perhaps more common than the instance in which they are all collinear. It is possible that for some other problem (say counting digits) there is no notion of "common in practice". Even in this case you can think of using *random* input values. You may wonder how you can feed *random* numbers to a computer. We will discuss this in Section 13.7.

Another possiblity is to consider if some input instances are "harder" than others, and hence might test the program better? The notion of *hard* is of course informal. But here is how you might consider certain inputs more *interesting*, say for the digit counting problem. If you look at the number of digits d as a function of the input n, you will see that d changes at powers of 10. At 9 the output value is 1, but it goes up to 2 at 10. The value is 3 at 999 but goes up to 4 at 1000. So you might want to pay more attention to these input values: perhaps the program has to be "keenly attentive" and distinguish between 999 and 1000 (even though they are consecutive), but not between 578 and 579 (which are also consecutive). So checking the inputs 999, 1000 might be more likely to show up the errors, than say checking 578 or 579. Another case of course is to check for the smallest and largest input values allowed. In case of digit counting 0 is the smallest value allowed, and whatever the largest value allowed is for representing n on your computer. The smallest, largest, and the values at which the output changes are informally called "corner cases", and you should certainly test around these values.

For the polygon-area problem, the simplest input instances could be rectangles, for which it should be easy to calculate the area by hand. You could again ask, what input instances are easy and which are hard? The polygons need to be simple, but of course they need not be convex. So if you plan to allow non-convex polygons as input, then certainly they should be a part of your test instances. If you decide that you don't want to allow non-convex instances, then you should amend the specification to declare this. Note that it is better that your program correctly implements a weaker specification than wrongly implementing a stronger one.

The length of the input is not fixed for the polygon-area problem. Very likely the program will first read in n, the number of vertices, and then the coordinates of the points. An important question to answer is how your program will handle corner cases, e.g. $n = 2$ or $n = 1$ or even $n = 0$. Either you should return 0, or you state clearly in the specification that these cases will not be handled by your program.

13.3 INPUT/OUTPUT REDIRECTION

When you write programs professionally, you are required to keep a record of the testing you have done. This can be done nicely by using a feature called *input redirection*. Most operating systems support input redirection.

In Chapter 1, we told you that `cin` represents the keyboard and whenever your program executes a statement of the form `cin >> ...` you are expected to type in an appropriate value. This is an oversimplification. If fact, `cin` represents an abstract, standard input device, which is the keyboard by default, but this default can be changed. If you wish, you can make the standard input device be a file, say named `file1`. Thus, instead of waiting for you to type in input, the program would take input from file `file1` whenever it executes a `cin >> ...` statement. This is called *input redirection*. To redirect input, you specify the name of the file on the command line, preceded by the character <, after the name of the executable. Thus to redirect input to come from `file1`, you will type the following on the command line.

```
% a.out <file1
```

As you can see, input redirection is very convenient. Even before you write the program, each test instances you create as discussed above can be placed in a file. When the program is ready, you run it, merely redirecting input so that the data comes from the file of your choosing.

Thus, we can suggest the following process for creating and using test cases. Even before you write the program create test cases, placing the input in files, one file per instance. Thus, you will create several files, say `input1.txt, input2.txt, ...` . Also create files `output1.txt, output2.txt, ...` which contain the outputs as you expect for the corresponding input instances. After the program is ready, simply redirect input, say from `input3.txt` in order to test it on the third instance you created. Check that the output you get is indeed what you had written down in `output3.txt`.

Note by the way that input redirection is also useful if your program does not run correctly on the very first run. If you have placed the data in a file then you can redirect input from it, and thus do not have to type the data again and again. This saves typing effort, and is especially useful for programs for which the input is large.

Note finally that the standard output stream, `cout` can also be redirected. For this you can execute your program by typing

```
% a.out >file2
```

If you do this, whatever you print by executing `cout << ...` in your program will be placed in the file `file2`, rather than being shown on the screen.

This is useful for programs which produce long output. If the output is put into a file, you can examine it at leisure. Also, the file thus created can serve as a record of your testing activities.

13.4 DESIGN OF ALGORITHMS AND PROGRAMS

The next step after the development of the specifications and test cases is *algorithm design*. By algorithm we mean the abstract ideas we need to solve a problem. For example, how do we find the smallest circle covering a set of points? This problem has a puzzle like flavour, and seems to require some creative thinking. You might even wonder whether creativity can be taught. But there do exist strategies for designing algorithms. As we have been mentioning frequently, recursion is one strategy. However, creative algorithm design in general is really outside the scope of this book.

For the most part, whatever algorithms you need to know in order to write programs described in the text and the exercises, we will either tell you directly, or they will be minor modifications of algorithms you somehow know already. And do realize that you know a lot of algorithms already. Indeed from childhood you have been learning a lot of algorithms, how to multiply two numbers, how to cook, how to ride a bicycle, and so on. You will need to express some of these algorithms using C++. This will not necessarily be easy. You may be executing the algorithms subconsciously, out of habit, but you will have to introspect on your actions and identify the patterns in them and express them in C++. This may be possible for some problems, e.g. multiplying one number with many digits by another number, but very difficult for others, e.g. deciding the next move in a game of chess. In any case, for the most part you should not need serious algorithm design, but you should certainly be able to introspect over skills you have learnt since childhood, verbalize them and express them in C++.

The next question is how to organize your program assuming you understand the specifications, have created the test cases, and know all the relevant mathematical/algorithmic ideas. This we call *program design*.

We remarked in Section 11.8.1 that any large program is best written by dividing it into small functions. Later we will see other ways of dividing programs into pieces, but the key point is that some such division will need to be there. Once you decide to divide a large program into pieces, we really need to apply the program design-process (recursively!!) to each piece. We must write out the specification for each piece (function), and design test cases for each function too. It is tempting to not test the individual functions, but to put together the entire program, and see if the whole thing behaves correctly. But if the whole thing does not behave correctly, it is tricky to figure out which function is not working right. So it is useful to first test each function separately. This means simply that if your program needs to calculate GCD very often, do write a GCD function, and then test it before you put it together with the rest of the program. Quite likely, this strategy, called *unit testing*, will save you time in debugging later.

The idea of writing specifications applies even in implementing a function itself. Say when you design a loop, you should be clear in your mind as to what the loop is intended to accomplish (specification), and be able to reason about it by writing invariants and a potential function. It is also a good idea to put these down as comments. Basically, these are all "defensive programming strategies" intended to minimize the chance of making mistakes.

Another important program strategy is as follows. Suppose you are writing a program which solves a somewhat complex problem. The natural plan might be to write a program to solve the entire problem in the very first attempt. Another idea is: consider a weaker specification first. Write the program to solve the weaker specification, testing it completely. Then try your hand at the original specification. The idea behind this weaker specification first strategy is as follows. You may think that you understand the grand specification. But often you may not, especially if you are inexperienced. As you try to implement the simpler specification you may realize the problem needs more thought. Thus it may be better to get on with writing code reasonably quickly—that experience will help you understand the difficulties.[1] The other alternative is to develop the full specification first and only then start writing the program, and start testing only after the entire program is ready. In this, you deny yourself the feedback (not to mention the satisfaction!) that you get from doing some testing. Because of the early feedback, the weak specification first approach might end up saving time and effort.

As an example consider the polygon-area problem. You may know Heron's formula for the area of a triangle given the lengths of the sides:

$$\text{Area } = \sqrt{s(s-a)(s-b)(s-c)}$$

where a, b, c are the lengths of the sides of the triangle, and $s = \frac{a+b+c}{2}$. Using this, you may consider it easier to calculate the area of a convex polygon, than that of non-convex one. So in this case, the *weak specification first* strategy would encourage you to first write the program for the convex case. However, the key point is that whatever you do, you should work to a specification, weak or strong. In other words, there must be truth in advertising!

[1] But do not begin coding unless you have at least the weak specification written down fully.

13.4.1 Mentally Execute the Program

Much of the advice being doled out in this chapter may be considered "obvious", and indeed it is. However, experience shows that human beings do not always take obvious precautions (e.g. wearing seatbelts in cars). So it is worth reiterating even obvious precautions and putting up a checklist.

Here is one such simple habit worth developing. After you finish writing a program and before you actually execute it, take a simple input instance and mentally execute your program if possible. This may be difficult for large programs, but it will help a lot while you are learning. For example, considering the digit counting program, you should mentally execute your program on some small input and satisfy yourself that it is correct. The mental execution will often alert you to some errors.

13.4.2 Test Cases for Code Coverage

We have already talked about designing test cases before the program is written. However, some additional test cases may be usefully designed after the program development finishes. The basic idea is: the test cases on which you run your code must exercise every piece of code that you wrote. This is important if your code has many `if` statements, say nested inside one another.

We will consider this issue in some exercises later, e.g. Exercise 2 of Chapter 16.

13.5 ASSERTIONS

The import of the previous discussion is: you should know your program well.

Your knowledge of the program is often of the form: "at this point in the program, I expect the value of this variable to be at least 0". Why not actually test and verify such expectations during execution? If your program is not running correctly, it might well be because something that you confidently expect is not actually happening.

C++ contains a facility which makes it easy to verify your expectations and produce error messages if the expectations are incorrect. Suppose you expect a certain condition `condition` to hold at a certain point in your program, you simply place the statement

```
assert(condition);
```

Here, `condition` must be a boolean expression. When control reaches this statement, `condition` is evaluated, and if it is false, the program halts with a message, typically stating "Assertion failed", and the line number and the name of the program file containing the assertion is also given. If the expression `condition` is true (as you expected), then nothing happens and the control just moves to the next statement.

To use `assert` you must include the line

```
#include <cassert>
```

at the top of your file.

For example, in the `gcd` function of Figure 9.1, you expect the parameter `n` to be positive. Thus, you could place the line

```
assert(n>0);
```

as the first line of the function body. If during execution, `gcd` is called with the second argument 0, you would get an error message saying that this assertion failed.

The preconditions of a function are natural candidates for being asserted. It may sometimes also be possible to assert that the function is indeed returning the correct value. For example, suppose we want to write a program to determine the integer part y of the square root of an integer x. The specification for this could be

> Find positive integer y such that $y^2 \leq x$ and $(y+1)^2 > x$.

In this case, the function could have the following form

```
int intsqrt(int x){
  assert(x >= 0);
  int y;

  ... code to compute y without changing x...

  assert(y >= 0 && y*y <= x && (y+1)*(y+1) > x);
  return y;
}
```

Just in case you made a mistake in the code, the assertion at the bottom would catch that and report it. That is better than returning a wrong value! However, this idea does not work very well in general. For example, how would you test whether the `gcd` function is indeed returning the GCD? The specification is that the value being returned should be the largest integer that divides both the arguments. However there is no easy way of checking this and converting to an assertion.

Here is another example. Suppose you know that a certain variable v will only take values 1 or 2. The you might originally have written:

```
...
if(v == 1) ...
else ...
...
```

Instead of relying confidently on your expectation, you have an opportunity to test it. For this you can write something as follows.

```
...
if(v == 1) ...
else if(v == 2) ...
else assert(false);  // executed only if v is not 1 or 2.
...
```

In this case, if `v` actually takes values besides 1 or 2, the assertion will fail and you will get a message.

We will see more examples of assertions later, e.g. for array bounds checking (Section 14.8).

13.5.1 Disabling Assertions

Assertions are meant to be used in the debugging phase, when you are not completely sure about the correctness of your program. Once you become sure of the correctness, you may want to remove the

assertions. This is because checking the assertions will take some time and will slow down the program. But it might be cumbersome to go through all the code and physically remove the assertions.

It turns out that is possible to disable assertions placed in your program without physically removing them. For this, you put in the line

```
#define NDEBUG
```

at the top of each file containing assertions you want to disable. The above line is to be placed before the inclusion of `<cassert>`. The above line will define the preprocessor (Appendix H) variable `NDEBUG`. This will have the effect of turning the `assert` statement into a comment.

13.6 DEBUGGING

Suppose you follow the above directions and are generally very careful, and yet things go wrong: your program produces an answer different from what you expect. What do you do?

Clearly, it would help if you knew the *earliest* point in the execution at which the program starts behaving differently from what you expect. For this, you can print out the values of the important variables at some convenient halfway point, and check if the values are as you might expect. If the values printed by these statements are as you expect (or not), then the error must be happening later (earlier), so you put print statements at later (earlier) points in your program. By examining proceeding in this manner, you can zoom in on the source of the problem. Presumably you will conclude something like "Everything is fine until statement xxx in the program is executed for the third time, but not after that". At this point, you are usually in a position to determine what is going wrong. The process of examining the values taken by variables during execution can be made much easier if you use programs called debuggers or IDEs. We discuss them below.

We do note one important source of errors: it might be the case that your program is not working correctly not because it has a logical flaw, but because it is not being fed the correct data. This can happen especially if the input is coming from a badly designed input file which you have redirected. We discuss how to deal with this.

13.6.1 Debuggers and IDEs

There exist specialized programs, called *debuggers* or *IDEs*, Interactive Development Environments, which are modern versions of debuggers, which can substantially help in the process of debugging.

Debuggers or IDEs offer many ways of executing your program. For example, you can ask that the program be *stepped*, i.e. run one statement at a time. You can see where the control is after the execution of the statement in question ends, and you can also examine the values of the different variables. You can also ask that the program execute until a certain statement is reached, executing freely till that statement. Once that statement is reached, you can again examine variable values if you wish. Essentially, this enables you to investigate how your program executes without having to put print statements in it.

Unfortunately, most IDEs are fairly complex, and it is significant work to just understand how to use them. That is the reason we have not discussed IDEs in this book. But if you plan to write programs with thousands of lines of code, you should learn to use IDEs.

13.6.2 End of File and Data-input Errors

C++ behaves in a somewhat unintuitive manner in data input. Suppose you execute `cin >> x;` where `x` is of type `int`. Suppose the value typed in response to this (or read from the file from which `cin` is redirected) happens to be the character 'a'. If this happens you might expect that the program will halt with an error message. However, the program does not halt! Instead, some junk value is supplied to you and the program continues merrily.

The simplest indication that an error has happened is that the value of `cin` becomes 0, or `NULL`. So ideally, after reading every value you should check if `cin` is not `NULL`. For this, you can write something such as

```
if(cin == NULL){cout << "Input error.\n"; exit(1);}
```

Or the shorter version:

```
if(!cin){cout << "Input error.\n"; exit(1);}
```

This is because `NULL` or 0 also stands for the logical value `false`.

The main point to note is as follows. Suppose your program is not working correctly. It could be because of a data input error. You may be feeding it an illegal value. This is not likely to happen if you are actually typing in values from the keyboard in response to messages from the program. However, if the program is reading data from a file (because of redirection or otherwise) it may well happen. So it is good to check for input errors. Also see Section F.2.

13.6.3 Aside: Input-Output Expressions

Finally, we note that in C++ the phrase `cin >> value` causes a value to be read into the variable `value`, and in addition itself is an expression that has a value: the value of the expression is the value of the variable `cin`. This should not come as a surprise to you, this is in fact the reason you can write statements such as `cin >> a >> b;` which you should really read as `(cin >> a) >> b;` where the first expression causes a value to be read into `a`, and then the expression evaluates to `cin`, from which another value is read into `b`.

This fact allows us to write some rather compact loops. Suppose you want to find the sum of a sequence of numbers stored in a file. You can do this by executing the following program with `cin` redirected from that file.

```
int main(){
  int val, sum=0;
  while(cin >> val){      // file read in the loop test
    sum += val;
  }
  cout << sum << endl;
}
```

The reading happens in the loop test, and if there is an error or end of file, the reading expression returns false, and the loop ends. Thus, the above loop will end when the file ends, and after that the sum will be printed.

Note that you can also use the above program to sum values that you type in from the keyboard. Just type in the values, and follow them with a ctrl-d (type 'd' while the control key is pressed), which signals an end of file.

13.7 RANDOM NUMBERS

C++ provides you with the function `rand` which takes no arguments and returns a *random* number. This statement should puzzle you—a computer is an orderly deterministic machine, indeed we did not say anything about randomness in our discussion of computer hardware (Chapter 2). How can then a computer generate random numbers?

Indeed, a computer does not generate truly random numbers. Instead, a computer merely generates successive numbers of a perfectly deterministically computed sequence whose elements *seem* to be resemble a sequence which could have been generated randomly. Such sequences and their elements are said to be *pseudo-random*. Indeed a simple example is the so called linear congruential sequence, given by $x_i = a \cdot x_{i-1} + b \bmod M$, where a, b, M are suitably chosen integers. Say we choose, just for the purpose of discussion, $a = 37$, $b = 43$, $M = 101$. Then starting with $x_0 = 10$, the next few terms are 9, 73, 17, 66, 61, 78, 0, 43, 18, 2, 16. Perhaps you will agree informally that this sequence looks random, or at least more random than the sequence 0, 1, 2, 3, 4 and so on. It is possible to formalize what *pseudo-random* means, but that is outside the scope of this book. So we will just assume that pseudo-random merely gets the best of both worlds: it is a sequence that can be generated by a computer, but can be considered to be random for practical purposes.

Functions such as `rand` which return (pseudo) random numbers do use the general idea described above: the next number to be returned is computed as a carefully chosen function of the previous. So the exact sequence of numbers that we get on successive calls to `rand` depends upon how we started off the sequence, what x_0 we chose in the example above. This first number of the sequence is often called the *seed*. C++ allows you to set x_0 to any value v you wish by calling another function `srand` which takes a single integer argument which you must specify as v. To use `rand` and `srand`, you would normally need to include the header file `<cstdlib>`. But this is included automatically if you include `<simplecpp>`.

A call `rand()` returns an `int` in the range 0 to `RAND_MAX`. This name is defined for you when `<cstdlib>` is included. You can consider the returned value to be *uniformly distributed*, i.e. the value is equally likely to be any integer between the specified limits.

Finally, an important point about pseudorandom sequences. The sequence you get when you fix the seed is always the same. This is a desirable property if you will use it to generate input data. This is for the following reason. Suppose your program is not working correctly for certain (randomly generated) data. Say you modify the program and you wish to check if it is now correct. Had the data been truly random, it would be unlikely that the same sequence would get generated during the execution. However, since you use a pseudo random sequence, you are guaranteed to get the same sequence if you set the same seed!

Of course, you might also want to the program to run differently on each occasion. In such cases, you can use the command `time` to set the seed, i.e. write

```
srand(time());
```

The `time` command returns the current time in seconds since some midnight of January 1, 1970, or some such moment. Clearly, `time()` will have a different value on each run. To use the `time` command, you must include the header file `<ctime>`.

13.7.1 The `randuv` Function in Simplecpp

In Simplecpp, we have provided the function `randuv` which takes two `double` arguments `u,v` and returns a random double in the range `u` through `v`. Our command calls the C++ supplied function `rand`, and returns the following value:

```
u + (v-u)*rand()/(1.0 + RAND_MAX)
```

As you can see this value will be between `u` and `v` and uniformly distributed to the extent `rand` is uniformly distributed.

If you want random numbers integers between given integers `i,j`, you must call `randuv(i,j+1)` and convert it to an integer. This will give you uniformly distributed integers between `i` and `j`.

You can use `srand` to set the seed as before.

13.8 CONCLUDING REMARKS

We began the chapter by stressing the need to clearly understand the specification; indeed many errors happen because the specifications are not properly understood by the programmer. We also discussed some strategies for developing test cases.

We discussed a few tools for helping the process of program development: input/output redirection, and assertions.

As to debugging, the main idea suggested was to put in print statements to see whether the program was executing as per your expectation. We also pointed out the possibility of errors in data input, and how to deal with them. As an aside we discussed the notion of input expressions, using which you can easily check if a file has ended.

We also discussed (pseudo) random number generation, which will be useful for generating random input instances. But (pseudo) random numbers are also useful in general, e.g. Chapter 27.

You may find many suggestions in this chapter to be very cautious, if not paranoid. But when it comes to serious programming, it is better in the long run to be humble and paranoid.

EXERCISES

1. For the digit-counting problem could the condition $10^d > n$ be $10^d \geq n$ instead? What if we did not require d to be a positive integer? Give a crisp answer, i.e. give inputs for which the new specifications would require an answer different (wrong!) from that required by the old one.
2. Here is a "clever" observation about the digit-counting problem. Suppose a number n has d digits. Then $\lfloor n/10 \rfloor$ has $d-1$ digits. Thus, we simply count the number of times we can divide by 10 till we get zero and that will be the number of digits of the number. So the program is

```
main_program{
  int n, d=0;
  cin >> n;

  while(n>0){
    n = n/10;
```

```
    ++d;
  }

  cout << "There are "<<d<<" digits.\n";
}
```

Is this program correct? Would you have written this program if you had followed the process suggested in this chapter? For what values of the input would you test the program?

3. Design input instances to test the income-tax-calculation problem of Chapter 6.

4. You are to find and correct the flaw in the following function. How will you go about it? State at least two strategies and see whether they work.

```
int gcd(int m, int n)
// finds GCD(m,n) for positive integers m,n
{
  if(m % n == 0) return n;
  else return gcd(m % n,n);
}
```

CHAPTER 14

Arrays

Here are some real-life problems that we may want to solve using computers.

- Given the marks obtained by students in a class, print out the marks in non-decreasing order, i.e. the smallest marks first.
- Given a roadmap of India, find the shortest path from Buldhana to Jhumri Telaiya.
- Given the positions, velocities and masses of stars, determine their state 1 million years from today.

In principle, we could write programs to solve these problems using what we have learned so far; however there will be some difficulties because of sheer size: our programs might have to deal with thousands of stars or hundreds of students or roads. Even writing out distinct names for variables to store data for each of these entities will be tiring.

Most programming languages provide convenient mechanisms using which we can tersely deal with large collections of objects. In C++ there are two such mechanisms.

Arrays: This is an older mechanism which was also present in the C language, and as a result can also be used in C++. In this chapter we will consider arrays at length.
Vectors: This is a newer mechanism, which is only present in C++. Vectors have essentially all the features of arrays, and some more.

We discuss arrays first because arrays are easier to understand, and because this discussion will be useful for understanding vectors later in Chapter 22.

14.1 ARRAY: COLLECTION OF VARIABLES

C++ allows us to write statements such as

```
int abc[1000];
```

This single statement defines 1000 variables! The first of these is referred to as `abc[0]`, the next as `abc[1]`, and so on till `abc[999]`. The collection of these 1000 variables is said to constitute the *array* named `abc`, and `abc[0]`, `abc[1]`, ..., `abc[999]` are said to constitue the *elements* of the array. Any identifier (Section 3.1.1) can be used to name an to array. What is inside `[ ]` is said to be the *index* of the corresponding element. The term *subscript* is also used instead of index. It is important

to note that indices start at 0, and not at 1 as you might be inclined to assume. The largest index is likewise one less than the total number of elements. The total number of elements (1000 in the above example) is referred to as the *length* or the *size* of the array. Assuming an `int` variable needs 4 bytes of space, the statement above reserves 4000 bytes of space in one stroke.

The space for an array is allocated contiguously in the memory of the computer, in the order of the index, i.e. `abc[1]` is stored in memory following `abc[0]`, `abc[2]` following `abc[1]`, and so on.

You may define arrays of other kinds also, e.g.

```
float b[500];  // array of 500 float elements.
```

You can mix up the definitions of ordinary variables and arrays, and also define several arrays in the same statement.

```
double c, x[10], y[20], z;
```

This statement defines variables `c, z` of type `double`, and two arrays `x, y` also of type `double`, respectively having lengths 10, 20. Assuming one variable of type `double` requires 8 bytes of space, this statement is reserving 8 bytes each for `c, z`, and respectively $8 \times 10, 8 \times 20$ bytes for `x,y`.

You may define arrays in the main program or inside functions as you wish. Note however, that variables defined inside functions are destroyed once the function returns. This applies to arrays defined in functions as well.

As per the C++ standard, the length of the array should be specified in the definition using a constant. However, also see Section 14.7.

14.1.1 Array-element Operations

Everything that can be done with a variable can be done with the elements of an array of the same type.

```
int a[1000];
cin >> a[0];  // reads from keyboard into a[0]

a[7] = 2;     // stores 2 in a[7].

int b = 5*a[7]; // b gets the value 10.

int d = gcd(a[0],a[7]); // gcd is a function as defined earlier.

a[b*2] = 234;   // index: arithmetic expression OK
```

In the first statement after the definition of `a`, we are reading into the zeroth element `a[0]` of `a`, just as we might read into any ordinary variable. You can also set the value of an array element by assigning to it, as in the statement `a[7]=2;`. The statement following that, `b=5*a[7];`, uses the element `a[7]` in an expression, just as you might use an ordinary variable. This is also perfectly fine. Note that just like ordinary variables, an element must have a value before it is used in an expression. In other words, it would be improper in the above code to write `int b = 5*a[8];` because `a[8]` has not been assigned a value.

Elements of an array behave like ordinary or *scalar* variables of the same type; so they can be passed to functions just like scalar variables. Hence, we can write `gcd(a[0],a[7]);` if we wish, assuming `gcd` is a function taking two *int* arguments.

In the last line in the code the index is not given directly as a number, but instead an expression is provided. This is acceptable. When the code is executed, the value of the expression will be computed and will be used as the index. In the present case, by looking at the preceding code we know that `b` will have the value 10, and hence `a[b*2]` is simply `a[20]`. So 234 will be stored in `a[20]`.

14.1.2 Acceptable Range for the Index

When using arrays in your programs, it is very important to keep in mind that the array index must always be between 0 (inclusive) and the array size (exclusive). For example, for the array `a` are defined above, a reference `a[1000]` would be incorrect, because it is not in the range 0 to 999. Likewise, a reference `a[b*200]` would also be incorrect, because it is really the reference `a[2000]` given that `b` has value 10 in the code above.

If such references are present in your program, its behaviour cannot be predicted. The program may generate wrong values, fail to terminate, or terminate with an error message. Any one of these outcomes is possible, and C++ does not say which will happen.

Simply put, it is vital that you, the programmer, make sure that array indices are in the required range. This is an extremely important requirement.

14.1.3 Initializing Arrays

It is possible to combine definition and initialization. Suppose we wish to create a 5 element `float` array called `pqr` containing respectively the numbers 15, 30, 12, 40, 17. We could do this as follows.

```
float pqr[5] = {15.0, 30.0, 12.0, 40.0, 17.0};
```

In fact, an alternate form is also allowed and you may write:

```
float pqr[] = {15.0, 30.0, 12.0, 40.0, 17.0};
```

in which the size of the array is not explicitly specified, and it is set by the compiler to the number of values given in the initializer list. You can of course mix definitions of arrays with or without initialization, and also the definition of variables.

```
int x, squares[5] = {0, 1, 4, 9, 16}, cubes[]={0, 1, 8, 27};
```

This will create a single `int` variable `x`, and two initialized arrays, `squares` of length 5, and `cubes` of length 4.

Of course, it might be more convenient to initialize arrays separately from their definitions, especially if they are large. So if we wanted a large table of squares, it might be more convenient to write

```
int squares[100];
for (int i=0; i<100; i++)
   squares[i] = i * i;
```

14.2 EXAMPLES OF USE

The common use of arrays is to store values of the same type, e.g. velocities of particles, marks obtained by students, lengths of roads, times at which trains leave, and so on. You could also say that an array is perfect to store any sequence $x_1, x_2, \ldots, x_n$. Of course, since array indices start at 0 in C++, it is more

convenient to call the sequence $x_0, x_1, \ldots, x_{n-1}$, and then store x_i in ith element of a length n array. As will be discussed in Section 15.1, an array can be used to store text. An array can also be used to store a machine language program: the ith element of the array storing the ith word of the program (Section 2.9). We will see many such uses in the rest of this chapter and the following chapters.

In this section we give some typical examples of programs that use arrays. You will see some standard programming idioms for dealing with arrays.

14.2.1 Notation for Sub-arrays

It will be convenient to have some notation to indicate sub-arrays of an array. Thus, we will use the notation `A[`i`..`j`]` to mean elements `A[`k`]` of the array `A` where $i \leq k$ and $k \leq j$. Note that if $i > j$, then the sub-array is empty.

This notation is only for convenience in discussions, it is not supported by C++ and cannot be used in programs.

14.2.2 A Marks-display Program

Suppose a teacher wants to announce the marks the students in a class have got. One way would be to put up a list on the school noticeboard. Another possibility is as follows. The teacher loads the marks onto a computer. Then any student that wants to know his marks types his roll number, and the computer displays the marks.[1] Can we write a program to do this?

For simplicity, let us assume that there are 60 students in the class, and their roll numbers are between 1 and 60. Let us also stipulate that the program must print out the marks of each student whose roll number is entered, until the value -1 is supplied as the roll number. At this point, the program must halt.

Clearly we should use an array to store the marks. It is natural to store the marks of the student with roll number 1 in the 0th element of the array, the marks of the student with roll number 2 in the element with index 1, and in general, the marks of the student with roll number i in the element at index $i - 1$. So we can define the array as follows.

```
float marks[60]; // marks[i] stores the marks of roll number i+1.
```

You are probably wondering whether we need to change the program if the number of students is different. Hold that thought for a while, we will discuss this issue in Section 14.7.

Next we read the marks into the appropriate array elements.

```
for(int i=0; i<60; i++){
  cout << "Marks for roll number " << i+1 << ": ";
  cin >> marks[i];
}
```

Remember that when the statement `cin >> marks[i];` is executed, the then current value of `i` is used to decide which element gets the value read. Thus, in the first iteration of the loop, `i` will have the value 0, and so what is read will be stored in `marks[0]`. In the second iteration, `i` will have the value 1 and so the newly read value will be stored in `marks[1]`, and so on. Thus indeed we will have the marks of a student with roll number `i+1` be stored in `marks[i]` as we want.

[1] Many might not like the idea of displaying marks in public. Later you can add a password so that each student can only see her marks.

In the last part of the program, students enter their roll numbers and we are to print out the marks for the entered roll number. Since this is to happen till -1 is given as the roll number, we clearly need a `while` loop. There are various ways to do this, we choose one with a `break`, similar to Section 7.2.

```
while(true){
  cout << "Roll number: ";
  int rollNo;
  cin >> rollNo;

  if(rollNo == -1) break;

  cout << "Marks: " << marks[rollNo-1] << endl;
}
```

Clearly, if you typed 35 in response to the query "Roll number: ", then you would want the marks for roll number 35, and these would be stored in `marks[34]`. But this is exactly the same element as what is printed, `marks[rollNo-1]`.

The program given above will work fine, so long as the roll number given is either -1 or in the range 1 through 60. If a number other than these is given, say 1000, the program will attempt to read `marks[999]`. As we said, this may result in some irrelevant data to be read, or worse, the program may actually halt with an error message. Halting is not acceptable in this situation, because students coming later will then not be able to know their marks. Fortunately, we can easily prevent this. If the roll number is not in the given range, then we can say so and not print any marks. So the code should really be as follows.

```
while(true){
  cout << "Roll number: ";
  int rollNo;
  cin >> rollNo;

  if(rollNo == -1) break;

  if(rollNo < 1 || rollNo > 60)
    cout << "Invalid roll number." << endl;
  else
    cout << "Marks: " << marks[rollNo-1] << endl;
}
```

14.2.3 Who Got the Highest?

Having read in the marks as above, suppose we wish to print out the roll numbers of the student(s) who got the highest marks, instead of answering student-marks queries.

What we want can be done in two steps. In the first step, we determine the maximum marks obtained. In the second, we print out the roll numbers of all who got the maximum marks.

In Section 3.4.1, we have already discussed how to find the maximum of the numbers read from the keyboard. Now instead of getting the marks from the keyboard, we are required to read them from the array. Basically, instead of reading from the keyboard, the first element will be obtained from

marks[0], and subsequent elements by looking at marks[i], where i has to go from 0 to 59. The code for this is as follows.

```
float maxSoFar = marks[0];
for(int i=1; i<60; i++){   // i starts at 1 because we already
                           // took marks[0]
  maxSoFar = max(maxSoFar, marks[i]);
}
```

The next step is to print the roll numbers of those students who got marks equal to maxSoFar. This is easily done, we examine each marks[i], for all i as i goes from 0 to 99, and whenever we find marks[i] equalling maxSoFar, we print out i+1, because we stored the marks of roll number i+1 at index i.

```
for(int i=0; i<60; i++)
   if(marks[i] == maxSoFar)
     cout << "Roll number " << i+1 << " got maximum marks." << endl;
```

14.2.4 General Roll Numbers

In the code above, we exploited the fact that the roll numbers are consecutive. In general this may not happen. Often, the roll number assigned to each student may encode different kinds of information, e.g. first two digits are year of joining, another digit indicates the department to which the student belongs, and so on. Sometimes the roll number may also contain letters, though for simplicity we will ignore this possibility.

We consider the marks display problem in this new setting. We will use an additional array rollno in which to store the roll number, in addition to the array marks used above. The teacher first types in 60 pairs of number, each pair consisting of a roll number and the marks obtained by the student having that roll number. Our program must read in the roll number and marks and store them in the arrays rollno and marks. In the second phase, when a student types in a roll number, we must first look for it in the array rollno. If it is found, then we print the corresponding marks.

```
int rollno[60];
double marks[60];

for(int i=0; i<60; i++) cin >> rollno[i] >> marks[i];

while(true){
  int r; cin >> r;  // roll number whose marks are requested
  if(r == -1) break;
  for(int i=0; i<60; i++)
     if(rollno[i] == r) cout << marks[i] << endl;
}
```

This idea, scanning an array from the beginning to the end in order to determine if a certain element is stored in the array, is sometimes called *linear search*.

The code above is unsatisfactory in two ways. First, if the given value r is not present in the array, it would be polite to print a message to that effect. Second, once we find r at some index, there is

no need to scan the remaining elements. Both these goals can be acheived by replacing the `for` loop above with the following.

```
int i;
for(i = 0; i<60; i++){
   if(rollno[i] == r){ cout << marks[i] << endl; break;}
}
if(i >= 60) cout << "Invalid roll number.\n";
```

Note first that we break out of the loop upon finding a match. Thus, if a match is found the variable `i` (which has now been defined outside the loop) will have a value less than 60. The check at the end succeeds only if all 60 iterations were executed without finding a match, i.e. if the roll number `r` is invalid.

14.2.5 Histogram

Our next example is trickier, and it illustrates an important powerful feature of arrays.

Again, we have as input the marks of students in a class. Assume for simplicity that the marks are in the range 0 through 99. We are required to report how many students got marks between 0 and 9, how many between 10 and 19, how many between 20 and 29, and so on. As you might know, what we are asked to report is often called a *histogram* in statistics[2].

We are required to report 10 numbers. So it could seem natural to use an array of 10 elements. The 0th element of the array can be used to count the number of marks in the range 0–9, the first element for the range 10–19, and so on. So in general we could say ith element of the array should correspond to the range `i*10` to `(i+1)*10-1` (both inclusive). So we call the array `count` and define it as:

```
int count[10]; // count[i] will store the number of marks in the
               // range i*10 through (i+1)*10 - 1.
```

Clearly, we should set the counts to 0 at the beginning, and change them as we read in the marks.

```
for(int i=0; i<10; i++)
  count[i]=0;
```

When we read the next mark, how do we decide which count to increment? It is natural to write something like the following.

```
  for(int i=0; i< 60; i++){          // can do better than this!
    float marks;
    cin >> marks;
    if(marks <= 9) count[0]++;
    else if(marks <= 19) count[1]++;
    else if(marks <= 29) count[2]++;
    else if(marks <= 39) count[3]++;
    else if(marks <= 49) count[4]++;
    else if(marks <= 59) count[5]++;
```

[2] In general, a histogram is a count of number of observations (marks, in our case) falling in various ranges of values (in our case the intervals 0–9, 10–19, and so on). The counts are often depicted as a bar chart, in which the height of the bars is proportional to the count and width to the size of the range.

```
  else if(marks <= 69) count[6]++;
  else if(marks <= 79) count[7]++;
  else if(marks <= 89) count[8]++;
  else if(marks <= 99) count[9]++;
  else cout << "Marks are out of range." << endl;
}
```

This works, but there is a better way! Suppose we read a mark m, which count should we increase? For this, we simply need to know the tens place digit of m. As you might observe, this is simply $\lfloor m/10 \rfloor$, i.e. the integer part of $m/10$. But we can get the integer part by storing into an integer variable! This is what the following code does.

```
for(int i=0; i< 60; i++){
  float marks;
  cin >> marks;
  int index = marks/10;
  if(index >= 0 && index <= 9) count[index]++;
  else cout << "Marks are out of range." << endl;
}
```

Note that this works only because all the ranges are of the same size. But this is very often the case when computing histograms.

14.2.6 A Taxi-dispatch Program

Suppose you are the Mumbai dispatcher for the Mumbai–Pune taxi service. Your job is as follows. Drivers of taxis that are willing to take passengers to Pune report to you and give you their driver ID number and wait. Passengers who want taxis also report to you. When a passenger reports, you check if there are any waiting taxis. If there are, you assign the taxi of the driver that reported to you the earliest. Clearly, once a taxi has been given to a passenger, you need not keep the corresponding ID number on your list. If no taxis are available, you let the passenger know. You are not expected to keep track of waiting passengers, though an exercise asks you to do precisely this. You may assume that at any given point there will not be more than 100 taxis waiting for passengers. You are to write a program which will help you dispatch taxis as required.

Let us make this more specific. Suppose that the dispatcher will type 'd' when a driver arrives, followed by the driverID. Likewise when a customer arrives, the dispatcher will type 'c', and expect the program to print the ID of the assigned driver. Finally, to terminate the program, we will have the dispatcher type 'x' (commonly used as abbreviation of eXit).

Next we decide what variables we might need and how we should be using them.

Clearly, we will need to store the IDs of the waiting drivers. It seems natural to use an array, say `driverID`, to store these. Assume for simplicity that the IDs are integers with 9 or fewer digits, i.e. that they will fit in `int`. The size of the array should equal the maximum number of drivers we expect will be waiting with us at any time. Most of the time there will be fewer drivers waiting with us than the size of the array, so we presumably need a variable `nWaiting` which will denote the number of waiting drivers.

We also need to somehow record the order in which the drivers arrived, because we want to assign the next customer to the driver who has registered with us the earliest. A natural way to do this is to store the earliest waiting driver at index 0, the next earliest at index 1, and so on. The ID of the driver that arrived last would be at index `nWaiting - 1`.

If a new driver arrives, we can store his ID at the index `nWaiting`, and increment `nWaiting`. If a customer arrives, we can assign the driver at `driverID[0]`. However, once we assign the driver, we must shift up all the other entries in the array, since we have decided that the waiting drivers must be stored starting at index 0. This is expressed in the following code.

```
const int n = 100; // estimate of max waiting drivers.
int driverID[n], nWaiting = 0;

while(true){  /* Invariants: nWaiting denotes the number of
                 waiting drivers. 0 <= nWaiting <= n.  IDs
                 of waiting drivers are in driverID, from
                 driverID[0] to driverID[nWaiting - 1] */
  char command; cin >> command;
  if(command == 'd'){                          // driver arrives
    if(nWaiting >= n) cout << "Queue full, try later.\n";
    else{
      cin >> driverID[nWaiting];
      nWaiting++;
    }
  }
  else if(command == 'c'){                     // customer arrives
    if(nWaiting == 0) cout << "Nothing available.  Try later.\n";
    else{
      cout << "Assigning " << driverID[0] << endl;
      for(int i=1; i < nWaiting; i++) // shift up waiting drivers
        driverID[i-1] = driverID[i];
      nWaiting--;
    }
  }
  else if(command == 'x') break;
  else cout << "Illegal command.\n";
}
```

Note that we have added checks to see if the array `driverID` is already full when a driver is to be entered, and to see if there is at least one element in it when a customer arrives.

You might think that perhaps there should be a way to write the program without having to shift up the entries in `driverID` when we assign the driver at index 0. Instead of moving up the drivers in the array, could we not adjust our notion about where the front of the queue is? Indeed this will work. But it will need a bit more care. To do this right, perhaps it is worth considering how we might have dispatched taxis without computers.

❑ Dispatching Without Computers

It is always worth thinking about how any problem, including taxi dispatching, might be solved without computers. Say the dispatcher writes the driverID numbers on a blackboard, top to bottom, as the drivers report. When a driver arrives, we put down the number at the bottom of the list. When a passenger comes in, the number at the top of the list is given to the passenger, and then the number is erased.

For simplicity, let us assume that our blackboard can only hold 100 driver ID numbers. Managing this space on the blackboard turns out to be slightly tricky. Suppose 60 drivers report, and you write down their numbers, starting at the top. Suppose you next have 50 passengers, so you match them to the top 50 numbers, which you erase. At this point, you have only 10 numbers on the board, however, they are not at the top of the board, but they start halfway down the board. Suppose now 60 more drivers report. You would place 40 of these numbers below the 10 you have on the board, and that would take you to the bottom of the board. Where should you place the remaining 20? It is natural to start writing numbers from the top again, as if the bottom of the board were joined to the top. Think of the blackboard as forming the curved surface of a cylinder! Thus, at this point, you have 70 numbers on the board. They begin at position 50 (the topmost position being 0), go to the last position, 99. Then they "wrap around" so that the last 20 numbers occupy positions 0 through 19 on the board. Positions 20-49 are then unused. This should not confuse us; say we make a mark next to the first waiting driver. When we assign a driver, we erase the number from the board, and also shift the mark down one. This works fine so long as the number of taxi drivers waiting at any time does not exceed 100.

❑ Emulating a Blackboard on a Computer

Our program will mirror the actions given above. We do not shift drivers up when a driver is assigned, instead we have a variable `front`, which will always contain the index of the element of `driverID` containing the earliest waiting unassigned driver. This performs the function of the mark that the dispatcher places on the board.

If there are `nWaiting` drivers waiting for customers, their IDs will appear at positions starting at `front`. We need to be slightly careful as we say this: how do we describe what happens when the board fills to the bottom and the dispatcher is forced to write the numbers starting at the top again? We would like to somehow say that index that comes "next" after the last index `n-1` (i.e. the "bottom" of the board) is index 0 (i.e. the "top" of the board). So instead of saying that the next index after `front` is `front+1`, we will say that it is `(front+1) % n`. If `front` has some value `i<n-1`, then `(front+1) % n` is just `front+1`. However, if `front` equals `n-1`, then `(front+1) % n` will indeed become 0 as we wish. Thus we will simply require that if there are `nWaiting` drivers that are waiting, their IDs will be at indices `front`, `(front + 1) % n`, ..., `(front + nWaiting - 1) % n`.

Next we consider what actions to execute when a driver arrives. As before, we accept the ID only if `driverID` is not full. The ID of the arriving driver must be added consistent with the property mentioned above, i.e. at position `(front + Waiting) % n`. After that we must increment `nWaiting`.

When a customer arrives, as before we first check if there are any waiting drivers. If there are, we assign the driver at the front of the queue, i.e. `driverID[front]`. We add one to `front` to get to the next element of `driverID`. however, since we want to consider the queue to start again from the top, the addition is done modulo `n`. Finally, we must decrement `nWaiting`.

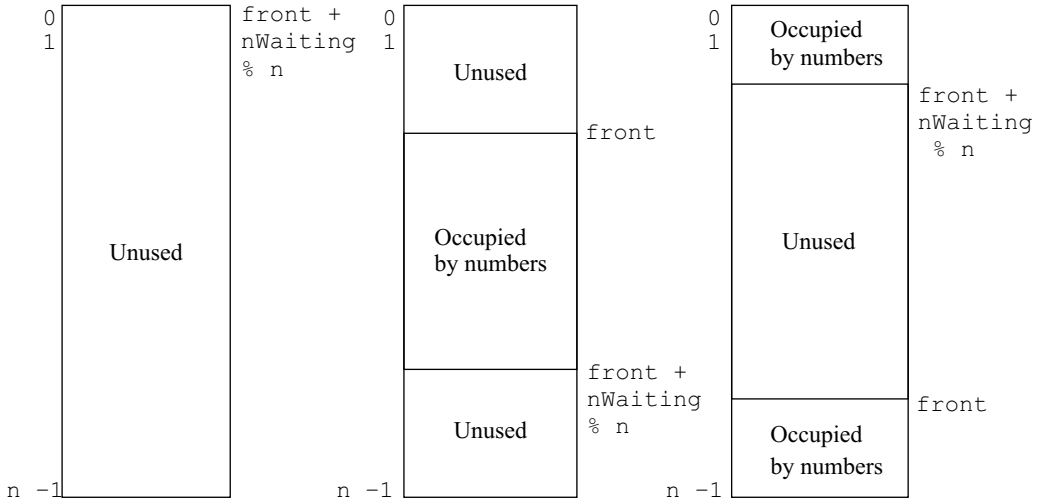


Fig. 14.1 *Snapshots of the board*

```
const int n = 100; // estimate of max waiting drivers.
int driverID[n], nWaiting = 0, front = 0;

while(true){ /* Invariants: nWaiting denotes the number of
                waiting drivers.
                0 <= nWaiting <= n.  IDs of waiting drivers are in
                driverID, from driverID[front] to driverID[(front +
                nWaiting - 1) %n ].
                0 <= front < n
             */
  char command; cin >> command;
  if(command == 'd'){                              // driver arrives
    if(nWaiting >= n) cout << "Queue full.  Try later.\n";
    else{
      cin >> driverID[(front + nWaiting) % n];
      nWaiting++;
    }
  }
  else if(command == 'c'){                        // customer arrives
    if(nWaiting == 0) cout << "Nothing available.  Try later.\n";
    else{
      cout << "Assigning " << driverID[front] << endl;
      front = (front + 1) % n;
      nWaiting--;
    }
  }
```

```
  else if(command == 'x') break;
  else cout << "Illegal command.\n";
}
```

You might have noted that it was easy to write this program once we decided clearly how our variables would be used. This is often a good strategy in writing programs.

14.2.7 A Geometric Problem

Suppose we are given the positions of the centers of several circles in the plane as well as their radii. Our goal is to determine whether any of the circles intersect. Let us say that the ith circle has center (x_i, y_i) and radius r_i, for $i = 0, \ldots, n-1$.

Whether a pair of circles intersect is easy to check: the circles intersect if and only if the distance between their centers is smaller than or equal to the sum of their radii. In other words, circle i and circle j intersect if and only if

$$\sqrt{(x_i - x_j)^2 + (y_i - y_j)^2} \leq r_i + r_j$$

Or equivalently, $(x_i - x_j)^2 + (y_i - y_j)^2 \leq (r_i + r_j)^2$. Thus, in our program we must effectively check whether this condition holds for any possible i, j, where of course $i \neq j$.

Here is how we can do this. We will use arrays `x, y, r` in which we will store the x, y coordinates of the center and the radius of the circles. Specifically, the x-coordinate of the center of the ith circle will be stored in `x[i]`, the y-coordinate in `y[i]`, and the radius in `r[i]`. We will then check whether each circle `i` intersects with a circle `j` where `j>i`.

```
int n=5;                // number of circles.  5 chosen arbitrarily.
float x[n], y[n];       // coordinates of center of each circle.
float r[n];             // radius of each circle.

for(int i=0;i<n;i++)    // read in all data.
  cin >> x[i] >> y[i] >> r[i];
                        // Find intersections if any.
for(int i=0; i<n; i++){
  for(int j=i+1; j<n; j++){
    if(pow(x[i]-x[j],2)+pow(y[i]-y[j],2) <= pow(r[i]+r[j],2))
      // built in function pow(x,y) = x raised to power y.
      cout << "Circles " << i << " and " << j << " intersect."
     <<endl;
  }
}
```

Thus, in the first iteration of the outer `for` loop, we check for intersections between circle 0 and $1, 2, 3, \ldots, n-1$. In the second iteration, we check for intersections between circle 1 and circles $2, 3, \ldots, n-1$, and so on. Is it clear that we check all pairs of circles in this process? Consider the kth circle and the lth circle, $k \neq l$. Can we be sure that the intersection between them is checked? Clearly, if $k < l$, then in the iteration of the outer for loop in which `i` takes the value k, we will check intersections with circles $k+1, k+2, \ldots, n-1$. This sequence will contain l because $k < l$. Alternatively, suppose $l < k$. Then consider the iteration of the outer `for` loop in which `i`$= l$. In this

iteration, we will check the intersection of circle l with circles $l+1, \ldots, n-1$. Clearly, k will be in this sequence because $l < k$. Thus in either case we will check the intersection between circle k and circle l, for every k, l.

14.3 THE INSIDE STORY

We now discuss some details regarding arrays and array accesses. This will specially be useful for understanding how we define functions for operating on arrays. To make the discussion more concrete, suppose we have the following definitions.

```
int p=5, q[5]={11,12,13,14,15}, r=9;
float s[10];
```

Say each variable of type `int` is given 4 bytes of memory, and so is a `float`. Thus, we know the above definitions will cause 4 bytes of memory to be reserved for `p`, $4 \times 5 = 20$ bytes for `q`, 4 for `r`, and $4 \times 10 = 40$ bytes for `s`. We have also said that the memory given for an array is contiguous. Thus, the memory for `q` will start at a certain address, say Q, and go on to address $Q + 19$. The notion of addresses is as per our discussion in Chapter 2 and Section 9.8. Consistent with this description, Figure 14.3 shows how space might have been allocated for these variables.

14.3.1 Time Required to Access an Array Element

Next we consider what happens when during execution we encounter a reference to an array element, e.g. `q[expression]`. How does the computer know where this element is stored? Of course, first the `expression` must be evaluated. Suppose its value is some v. Then we know that we want the element of `q` of index v. But because the elements are stored in order, we also know that the element with index v is stored at $Q + 4v$, where Q is the starting address for `q`. Thus, if $v = 3$ then we would want `q[3]`, which is stored from $Q + 12$. In general, the vth element of an array which is stored starting at address A would be at $A + kv$, where k is the number of bytes needed to store a single element. Indeed, to access an array element, the computer must evaluate the index expression, and then perform the multiplication and addition to get the address $A + kv$. In contrast, the address of an ordinary variable such as `p` is directly known, and no calculation needs to be performed.

So two points are to be noted. First, accessing an array element takes somewhat longer than accessing a simple variable. Second, and this is perhaps more important: the time required to access an element does not depend upon the size of the array.

14.3.2 Out-of-range Array Indices

Suppose now that our program has a statement `q[5]=17;`. Using the formula $A + kv$ given above, the computer would try to store 17 in the `int` beginning at the address $Q + 4 \times 5 = Q + 20$. Notice that this is outside the range of memory allocated for q. In fact, it is quite possible, as shown in our layout of Figure 14.3, that `r` is given the memory $Q + 20$ through $Q + 23$. Then the statement `q[5]=17` might end up changing `r`! Likewise it is conceivable that a statement like `q[-1]=30;` might end up changing `p`.

Suppose on the other hand, we wrote `q[10000]=18;`. This would require us to access address $Q + 40000$. It is conceivable that there isn't any memory at this address. Many computers have some circuits to sense if an access is made to a non-existent address or even some forbidden addresses. The details of this are outside the scope of this book, but if this happens, then the program might halt with

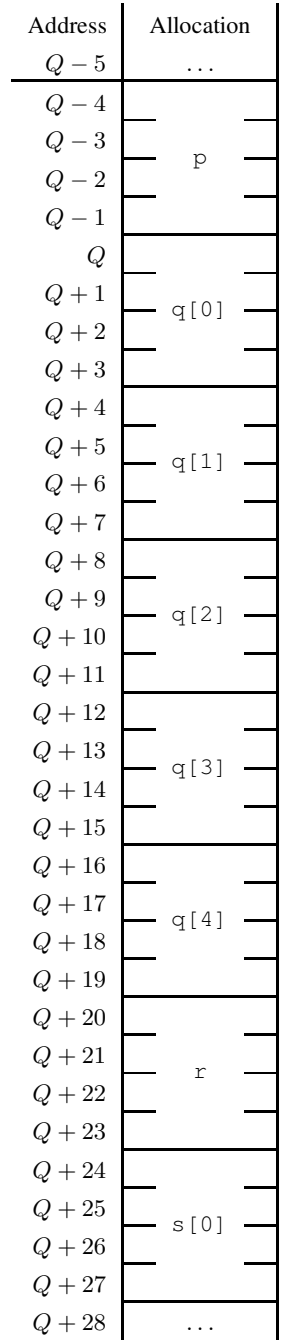


Fig. 14.2 *Possible placement of* `p,q,r,s` *in memory*

an error message. In any case, it is most important to ensure that array indices are within the required range.

14.3.3 The Array Name by Itself

So far we have not said whether the name of an array can be used in the program by itself, i.e. without specifying the index. It turns out that C++ allows this.

In C++, the name of an array by itself is defined to have the value equal to the starting address from where the array is stored. This is an important place where arrays work differently from vectors, as we will see in Chapter 22.

Continuing our example of Figure 14.3, since the array `q` is stored starting at address Q, the value of the name `q` itself would thus be Q, and the value of `s`, $Q + 24$. Since the variable at address Q is `q[0]`, of type `int`, it is natural to define the type of `q` to be pointer to `int`, or address of `int`, or `int*`. In general, if an array contains elements of type `T`, then its name will have type `T*` or address of `T` or pointer to `T`.

Thus, `s` would be of type address of `float` or pointer to `float` or `float*`, and would have the value $Q + 24$.

It seems strange that the name of an array is only associated with the starting address, and that the length of the array is not associated with the name. This is merely a matter of convenience, and its utility will become clear in Section 14.4.

An important point to note is that the value associated with the name of an array, say `q`, cannot be changed; it always means the address of `q[0]`. In other words, you cannot write an expression such as

```
q = ...;  // incorrect if q is the name of an array
```

Such expressions will be flagged as errors by the compiler.

14.3.4 The Operator "`[]`"

A further tricky point is that when you refer to an array by writing something like `X[Y]`, C++ considers it to be an expression, with `X, Y` the operands, and `[]` the operator![3] The operation is defined only if `X` has the type "address of some type `T`", and `Y` is an expression that evaluates to a value of type `int`. Suppose that `X` is of type address of type `T`, and `Y` does evaluate to `int`. Then the expression `X[Y]` denotes the variable of type `T` stored at the address $A + kv$, where A is the value of `X`, v the value of `Y`, and k is the number of bytes needed to store a single element of type `T`.

You will realize that we are merely restating how we find the element given the name of the array and the index. But the restatement is more general: `X` does not need to be the name of an array, it could be any name whose type is "address of some type `T`". This generalization will come in useful in Section 14.4.

Note by the way that `[]` will be written even when we define an array; in such contexts of course the `[]` is not to be considered an operator.

Just to drive home the point, consider the following code.

```
double speed[]={1.25, 3.75, 4.3, 9.2};  // [] is not an operator
double *s;
```

[3] Yes, this is an unusual way of writing a binary expression. But do note that there are other operations which are not written in the order operand1 operator operand2. For example, we often write $\frac{a}{b}$ rather than $a \div b$.

```
s = speed;
cout << s[0] << endl;                    // [] is an operator
s[1] = 3.9;
cout << speed[1] << endl;                // [] is an operator
```

The first point to note is that the assignment `s = speed` is very much legal, since `speed` is of type `double*`, just like `s`. Thus, after the assignment, `s` will have the same value as `speed`. But then, the expressions `s[j]` will mean the same as `speed[j]`.

Put differently, the expression `s[0]` denotes the double value stored at the address `s + k*i`, where `k` is the size of `double`, and `i` the value of index, which are respectively 8 and 0. In other words, s[0] means the double stored at address `s` which is the same as `speed`, and hence is the same as the value stored at address `speed`, and so is `speed[0]`. Thus, the first print statement will print 1.25. The statement `s[1]` is likewise equivalent to `speed[1]`, i.e. to 3.9, which is what the second print statement will print.

14.4 FUNCTION CALLS INVOLVING ARRAYS

Functions are convenient with ordinary, or *scalar* variables, and indeed we can imagine that they will be convenient with arrays as well. Suppose we have an array of floats defined as `float a[5];` and somewhere in the program we need to calculate the sum of its elements. It is not difficult to write the code to compute the sum of the elements of an array, however, if the sum is needed for several such arrays in our code, then will have to replicate the code that many times. So it would be very convenient to write a function which takes the array as the argument and returns the sum.

As it happens, we have told you everything you need to write the function! Here is what you could write.

```
float sum(float* v, int n){  // function to sum the elements
                             // of an array
  float s = 0;
  for(int i=0; i<n; i++)
    s += v[i];

  return s;
}
```

We will explain this shortly. First, we show how this function might be called from a main program.

```
int main(){
  float a[5] = {1.0, 2.0, 3.0, 4.0, 5.0},  asum;
  asum = sum(a, 5);
}
```

Let us first check whether the function call is legal, i.e. whether the types of the arguments match those of the parameters in the definition in the function `sum`. The first argument to the function call is the array name `a`. We said that the type associated with the array name is `T*`, if the elements in the array are of type `T`. Thus, the type of `a` is `float*`. This indeed matches the type of the first parameter, `v`, in the function definition. The second argument, 5, clearly has type `int` which matches the type of the second parameter `n`. Thus, the call is legal and we now think about how it executes.

When the call `length(a, 5)` is encountered, as usual an area is created for execution of the function `sum`. The values of the non-reference arguments are copied. In the present case, none of the parameters are reference parameters, and so the values of both arguments are copied. The value of the first argument, `a`, is the starting address, say A, of the array `a` in memory. Thus, `v` gets the value A. The value of the second argument is 5. Thus, `n` gets the value 5. It is very important to note here that the content of all the locations in which the array is stored are not copied, but only the starting address is copied.

The code of the function is then executed. The only new part is the expression `v[i]`. This is processed essentially according to the rule given earlier. We know that `v` has type address of float, and its value is A. So now the expression `v[i]` is evaluated as discussed in the previous section, by considering `[]` to be an operator and so on. Instead of doing the precise calculation again, we merely note that the value of `v[i]` evaluated in `sum` must be the same as the value of `a[i]` evaluated in the main program, because `v` has the same value and type as `a`. Hence, `v[i]` will in fact denote the `i`th element of `a`. Because `n` has value 5, in the loop `i` will take values from 0 to 4. Thus, `a[0]` through `a[4]` will be added as we desired.

Some remarks are in order.

1. Another syntax can also be used to declare parameters that are arrays, in this case arrays of `float` variables: `float v[]`—this directly suggests that `v` is like an array except that we do not know its length.
2. Function `sum` does not really *know* that it is operating on the entire array. For example the call `sum(a,3)` is also allowed. This would return the sum of the first 3 elements of `a`, since the loop in the function will execute from 0 to 2.
3. Modifying the passed array is also possible. If your function had a line at the end such as `v[0]=5.0;`, that would indeed change `a[0]`. This is consistent with the mechanism we have discussed for evaluating expressions involving `[]`.

14.4.1 Examples

Shown below are two simple examples of functions on arrays. The next sections gives more involved examples.

Our first function merely prints the values of the elements of a `float` array.

```
void print(float *a, int n){  // or: "void print(float a[], int n){"
  for(int i=0; i < n; i++)
    cout << a[i] << endl;
}
```

This will print out the first `n` elements of the array. Note that it is the responsibility of the calling program to ensure that an array is passed, and that the array has length at least as much as the second argument.

Next, we present a function which returns the index of an element whose value is the maximum of all the elements in the array. Note the careful phrasing of the last sentence: when we say "*an* element", we acknowledge the possibility that there could be many such elements, and we are returning the index of only one of them.

The idea of the function is very similar to what we did for finding the maximum marks from the `marks` array in Section 14.2.3. We will scan the array from left to right (by which we will mean from

index 0 to the largest address). We have a variable `maxIndex` in which will store the index of the largest element seen so far in the scan. We start by initializing it to 0, which is equivalent to aserting that the maximum appears in position 0 as we start the scan. Next, we check if the subsequent elements of the array are larger, if we find an element which is larger, then we assign its index to `maxIndex`.

```
int argmax(float marks[], int L)
// marks = array containing the values
// L = length of marks array.  required > 0.
// returns maxIndex such that marks[maxIndex] is largest in
// marks[0..L-1]
{
  int maxIndex = 0;
  for(int j = 1; j < L; j++)
    if( marks[maxIndex] < marks[j]) // bigger element found?
      maxIndex = j;                 // update maxIndex.
  return maxIndex;
}
```

We have given the name `marks` so that it is easy for you to see the similarity between this code and the code in Section 14.2.3. But by itself this function does not have anything to do with marks. So if you write it independently some more appropriate name such as `datavalues` should be used instead of the name `marks`.

14.4.2 Summary

The most important points to note are as follows.

To pass an array to a function, we must typically pass 2 arguments, the name of the array, and the length of the array. This is to be expected, the name only gives the starting address of the array, it does not say how long the array is. So the array length is needed.

The called function can read or write into the array whose name is sent to it. This is like sending the address of one friend A to another friend B, Surely then B will be able to write to A or visit A just as you can!

Finally, it is worth noting an important point. When we write a function on arrays, it may be convenient to allow it to be called with length specified as 0. What should a function such as `sum` to when presented with an array of zero length? It would seem natural to return the sum of elements as 0. This is what our `sum` function does. On the other hand, our `argmax` function requires that the length be at least 1. Such (pre)conditions on acceptable values of parameters should be clearly stated in the comments.

14.5 SELECTION SORT

We will consider a problem discussed at the beginning of the chapter: given the list of marks, print them out in the order lowest to highest. We could ask that along with the marks, we also print out the roll numbers, however, this is left for the exercises.

We can accomplish our task in two phases. In the first phase, we rearrange the element values in a non-decreasing order, i.e. so that the values appearing in element with smaller indices are no larger

than those appearing in element with larger indices. This operation is often called *sorting*. This is one of the most important operations associated with an array. We will present a simple algorithm called *Selection sort* for this. Better ways will be given in Section 16.2 and Section 22.3.2. Once the elements are arranged in a non-decreasing order, we can simply print out the array elements by index, i.e. element 0, then element 1, and so on. This will ensure that the marks are printed in non-decreasing order. For this, we can simply use the function `print` defined earlier.

We use a fairly natural idea for sorting. We begin by looking for the largest value in the array, and we move it to the last position, i.e. index $n-1$, where n is the length of the array. Of course, position $n-1$ itself contains a value, and we cannot destroy that. So we instead exchange the two: the maximum value moves to the $n-1$th position and the value in the $n-1$th position moves to wherever the maximum was present earlier. Next, we find the maximum value amongst elements in positions 0 through $n-2$. This maximum is exchanged with the element in position $n-2$. Thus, we have the maximum and second maximum at positions $n-1$ and $n-2$. In general, we proceed in this manner, in a typical iteration, we will find the maximum from the first `i` values, and then exchange that with the value at the `i-1`th index. We will have `i` begin with the value `n`, and count it down in successive iterations till we reach 2.

For finding the maximum, it is convenient to use the `argmax` function defined in Section 14.4.1. If we want the maximum from the first `i` elements, we simply invoke it using `i` as the second argument. As we noted there, `argmax` need not be passed the *actual* length of the array; if it is passed a smaller value `i` it will merely find a maximum in the first elements and return its index. So the code is quite obvious.

```
void SelSort(float data[], int n)
// will sort in NON-DECREASING order.
{
  for(int i=n; i>1; i--){
    int maxIndex = argmax(data,i);  // Find index of max in
                                    // data[0..i-1]
    float maxVal = data[maxIndex];  // Exchange elements at
    data[maxIndex] = data[i-1];     // index = maxindex
    data[i-1] = maxVal;             // and index = i-1.
  }
}
```

It should be clear that the code above is doing what we described. It is instructive to write a loop invariant also: At the beginning of each iteration, the subarray `data[i..n-1]` contains largest values, for the then value of `i`. Note that in the very first iteration `i = n` and hence the invariant is vacuously true.

Suppose the invariant is true at the beginning of some iteration, we will prove that it will also hold for the next. The first statement of the loop finds the maximum in the first `i` elements, i.e. elements 0 through $i-1$. The last 3 statements exchange this with the element at index $i-1$. Thus, at the end of the iteration, the subarray `data[i-1..n-1]` will contain the largest values. This establishes the invariant for the beginning of the next iteration.

14.5.1 Estimate of Time Taken

We will try to get a rough estimate of the time needed by Selection sort. By *rough estimate* we merely mean whether the time is proportional to n, the number of elements being sorted, or their square and so on. Such estimates cannot be used to decide how many seconds will be needed to execute the program. However, if we know that one program sorts in time proportional to n, and another in time proportional to n^2, then for large enough n, the first algorithm will be better. Usually, we care about the time taken only when the problem size, n in this case, is large. So our rough analysis: whether the time is proportional to n or n^2, is quite useful.

To analyze the time for `SelSort`, we must first analyze the time for `argmax`. The function `argmax` simply goes over the subarray on which it is called and finds the maximum. It examines every element and thus its time can be considered to be proportional to `L`, the second argument. In selection sort, we merely call `argmax` several times, with the value of the second argument being $n, n-1, n-2$ and so on till 2. Thus, we can see that the time is proportional to

$$n + (n-1) + (n-2) + \ldots + 2 = \frac{(n+2)(n-1)}{2} \approx n^2/2$$

Thus we estimate the time taken by Selection sort as being proportional to n^2, where n is the length of the array being sorted.

14.6 REPRESENTING POLYNOMIALS

A program will deal with real life objects such as stars, or roads, or a collection of circles. It might also deal with mathematical objects such as polynomials. How to represent polynomials on a computer and perform operations on them are therefore important questions.

A polynomial $A(x) = \sum_{i=0}^{i=n-1} a_i x^i$ is completely determined if we specify the coefficients $a_0, \ldots, a_{n-1}$. Thus, to represent the above polynomial we will need to store these coefficients. This most conveniently done in an array.[4] We use an array `a` of length n and store a_i in `a[i]`.

Next comes the question of how we operate on polynomials. It is natural to ask; suppose we have two arrays representing two polynomials $A(x), B(x)$. Can we construct the representation of the polynomial $C(x)$ that is the sum of polynomials $A(x), B(x)$, and likewise the polynomial $D(x)$, the product?

First, we know that the sum will have degree $n-1$ because the addends $A(x), B(x)$ have degree $n-1$. Thus, the array `c` that we can use to represent the polynomial $C(x)$ must also have length n. We also know that $c_i = a_i + b_i$. Thus, we know how to set the elements of the array `c` as well.

```
for(int i=0; i<n; i++)  c[i] = a[i] + b[i];
```

Can we write this as a function `addp` which adds two polynomials? The polynomials to be added will be passed as arguments. What about the result polynomial? We could allocate a new array inside the function `addp`, but this array cannot be returned back—it gets destroyed as soon as `addp` finishes execution. The correct way to write this procedure is to pass the result array as well. Here is our code written as a function `addp`.

[4] There is a simple rule here—if a collection of objects is described using one subscript, use a one dimensional array, which is what we have studied so far. If a collection of mathematical object is described using two subscripts, say the entries of a matrix, then we will need two dimensional arrays, which we will see later.

```
void addp(float a[], float b[], float c[], int n){
// a, b: addends, c: result, n: length of the arrays.
  for(int i=0; i<n; i++)  c[i] = a[i] + b[i];
}
```

We have assumed in the above program that the addend polynomials have the same degree. This need not be the case in general. But you should be able to modify the code to handle the general case.

We next consider the problem of computing the product polynomial $D(x)$ of our polynomials $A(x), B(x)$. As before, we will assume that both $A(x), B(x)$ have degree $n-1$, and are stored in arrays `a, b` of length n. The product will have degree $2n-2$, and must therefore be stored in an array `d` of length at least $2n-1$.

To determine $D(x)$, consider how its coefficients relate to those of $A(x), B(x)$. When $A(x)$ and $B(x)$ are multiplied, each term $a_j x^j$ in the former will be multiplied with $b_k x^k$ in the latter, producing terms $a_j b_k x^{j+k}$. Thus, this will contribute $a_j b_k$ to d_{j+k}. Thus we start by setting every coefficient of D to 0, and then for all j, k compute $a_j b_k$ and add it the coefficient d_{j+k}. This gives us the function.

```
void prodp(float a[], float b[], float d[], int n){
// a,b must have n elements, product d must have 2n-1.
  for(int i=0; i<2*n-1; i++) d[i] = 0;
  for(int j=0; j<n; j++)
    for(int k=0; k<n; k++)
       d[j+k] += a[j]*b[k];
}
```

To complete the example, here is a main program which calls these functions.

```
int main(){
  float a[5], b[5], c[5], d[9];
  for(int i=0; i<5; i++) cin >> a[i];
  for(int i=0; i<5; i++) cin >> b[i];

  addp(a,b,c,5);
  prodp(a,b,d,5);

  for(int i=0; i<5; i++) cout << c[i] <<' ';
  cout << endl;
  for(int i=0; i<9; i++) cout << d[i] <<' ';
  cout << endl;
}
```

14.7 ARRAY LENGTH AND `const` VALUES

In the examples given above, we have explicitly written out numbers such as 500,1000 to specify the array length. Arrays will often be used in programs for storing a collection of values, and the total number of values in the collection will not be known to the programmer. So you might consider it more convenient if we are allowed to write

```
int n;
cin >> n;
int a[n];  // Not allowed by the C++ standard.  But read on!
```

This code is not allowed by the C++ standard. The C++ standard requires that the length be specified by an expression whose value is a *compile time constant*. A compile time constant is either an explicitly stated number; or it is an expression only involving variables which are defined to be `const`, e.g.

```
const int n = 1000;
```

The prefix `const` is used to say that `n` looks like a variable, and it can be used in all places that a variable can be used, but really its value cannot be changed. So using a `const` name, arrays might be defined as follows.

```
const int NMAX = 1000; // convention to capitalize constant names.
int a[NMAX], b[NMAX];
```

So how do we use this in practice? Suppose we want to define an array which will store the marks of students. In this case, the C++ standard will require us to guess the maximum number of students we are likely to ever have, define an array of that size, and only use a part of it. So we might write

```
const int NMAX = 1000;
int a[NMAX], b[NMAX], nactual;
cin >> nactual;
assert(nactual <= NMAX);
```

In the rest of the code, we remember that only the first `nactual` locations of `a` and `b` are used, and so write loops keeping this in mind. Note that it is possible that the user will type in a value for `nactual` that is larger than `NMAX`. In this case we cannot run the program. If this happens, the `assert` statement will cause the program to stop, and you will need to change `NMAX`, recompile and rerun.

14.7.1 Why `const` Declarations?

The above code could also directly define `int a[1000],b[1000];` instead of using the `const` definition. However, the code as given is preferable if we ever have to change the required size, say we want arrays of size 2000 rather than 1000. If we had not used `NMAX` we would have to change several occurrences of 1000 to 2000; with the code as given, we just need to change the first line to `const int NMAX = 2000;`

14.7.2 What We Use in This Book

The GNU C++ compiler that you invoke when you use the command `s++` allows arbitrary expressions to be specified as length in an array definition.

As you can see, this makes the code much more compact and easier to understand at a glance. So in the interest of avoiding clutter, in the rest of the book, we will use arbitrary expressions while specifying lengths of arrays. The code we give will work with `s++`. If it does not work for some other compiler, the discussion above tells you how to change it.

14.8 CONCLUDING REMARKS

Arrays provide an easy way to store sets of objects of the same type.

It is worth thinking about how the index of an element gets used. Sometimes the index at which an element is stored has no significance, as in the circle intersection problem. Or sometimes we can make a part of the data be the index, as we did for the roll number in the marks display problem. Similar was the case for the histogram problem. In the taxi dispatch problem, we used the index to implicitly record the arrival order of the taxis.

Suppose we want to look for elements satisfying a certain property. One way to do so is to scan through the array, one element at a time, and check if the element has the required property. We did this in the problem of printing roll numbers of students who had the highest marks. This is a common idiom.

The idea of scanning through the array starting at index 0 and going on to the largest index is also useful when we want to perform the same operation on every element, e.g. print it. We used a somewhat complicated version of this in the circle intersection problem, where we wanted to perform a certain action not for each circle, but for each pair of circles.

In the taxi dispatch problem we built a so called *queue* so that the elements left the array in the same order that they arrived in. For this, we maintained two indices: where the next element will be stored and which element will leave next. This is a very common idiom.

Remember that the index used for an array should be in range, i.e. between 0 (inclusive) and the array length (exclusive). Having the index out of range is a common cause of errors in programs involving arrays. So you should make sure that the index is in the range. You could also consider checking this using assertions. For example, if you have an array `x` of length 200, which you are about to index using an index `i`, you could consider placing an assertion:

```
assert((i >= 0) && (i < 200));
```

before writing `x[i]`.

Finally, we considered how to write functions involving arrays. The important point to note here is that the array name can be used in programs, and by itself it denotes the starting address of the region of memory that holds the array. The length of the array must also be passed as an argument to the function.

EXERCISES

1. A sequence $x_0, \ldots, x_{n-1}$ is said to be a palindrome if $x_i = x_{n-1-i}$ for all i. Write a program which takes as input an integer n and then a sequence of length n and determines whether the sequence is a palindrome.
2. Write the program to display who got the maximum marks for the case when the roll numbers are arbitrary integers, as in Section 14.2.4.
3. Suppose we want to find a histogram for which the widths of the intervals for which we want the counts are not uniform. Say each value is a real number between 0 (inclusive) and 1 (exclusive). Between 0 and 0.25, our intervals are of width 0.05, i.e. we want a count of how many values are between 0 and 0.05, then 0.05 and 0.1, and so on. Between 0.25 and 0.75 our intervals are of width 0.025, i.e. we want to know how many values are between 0.25 and 0.275, then 0.275 and 0.3, and so on. Finally, between 0.75 and 1, our intervals are of width 0.05. Write a program that provides the histogram for these ranges.
4. List out the likely errors in the following program.

```
int main(){
   int x[10], y[100], i = 50, j;
   for(int k=1; k<= 10; k++){
      x[k] = k*k;
      y[k] = k*k*k;
   }
   cout << y[i] <<' '<< y[j] << endl;
}
```

You do not know what the program is expected to do, and clearly it is not doing much. Nevertheless, report whatever you think is likely to be erroneous. For example, a print statement likely to print different things in different runs is clearly erroneous.

5. Write a program which takes as input a sequence of positive integers, and prints the 10 largest numbers in the sequence. You are not given the length of the sequence before hand, but after all the elements of the sequence are given as input, the number -1 is given, to indicate that the sequence has terminated. *Hint*: Use an array of length 10 to keep track of the numbers that are candidates for being the top 10.

6. Suppose in the previous problem, you are asked to report which are the 10 highest values in the sequence, and how frequently they appear. Write a program which does this. Assume the sequence has length 1000.

7. Suppose we are given the x, y coordinates of n points in the plane. We wish to know if any 3 of them are collinear. Write a program which determines this. Make sure that you consider every possible 3 points to test this, and that you test every such triple only once. The coordinates should be represented as `floats`. When you calculate slopes of line segments, because of the floating point format, there will be round-off errors. So instead of asking whether two slopes are equal, using the operator ==, you should check if they are approximately equal, i.e. whether their absolute difference is small, say 10^{-5}. This is a precaution you need to take when comparing floating point numbers. In fact, you should also ask yourself whether the slope is a good measure to check collinearity, or whether you should instead consider the angle, i.e. the arctangent of the slope. Also write the program for the case that the coordinates are given as `int`.

8. Write a program that finds the smallest circle covering a given set of points. Allow the user to supply the points by clicking on the screen, and show the smallest circle also on the screen. *Hint*: Argue that the smallest covering circle must either have as diameter some two input points, or must be a circumcircle of some three input points. Now just consider all possible candidate circles, and pick the one that actually covers all points.

9. Write a program which takes as input two vectors (as defined in mathematics/physics), represents them using arrays, and prints their dot product. Make this into a function.

10. Suppose you are given the number n of students in a class, and their marks in two subjects. Your goal is to calculate the correlation. Let x_i, y_i denote the marks in the two subjects. Then the correlation is defined as

$$\frac{n\sum x_i y_i - \sum x_i \sum y_i}{\sqrt{n\sum x_i^2 - (\sum x_i)^2}\sqrt{n\sum y_i^2 - (\sum y_i)^2}}$$

Write a program that calculates this. Note that a positive correlation indicates that x increases with y (roughly), whereas negative correlation indicates that x increases roughly as y decreases.

A correlation around 0 will indicate in this case (and often in general) that the two variables are independent. You may use the dot product function you wrote for the previous exercise.

11. Suppose you are given the maximum temperature registered in Mumbai on 21 March of each year for the last 100 years. You would like to know whether Mumbai has been getting warmer over the years, as is generally believed. You would like to know from your data whether this might be a reasonable conclusion. If you merely plot the data, you will see that the temperatures fluctuate apparently erratically from year to year. The weather is expected to behave somewhat randomly; what you want to know is whether there is any upward trend if you can somehow throw out the randomness.

One way to reduce the local randomness is to smooth the data by taking so called *moving averages*. Given a sequence of numbers $x_1, \ldots, x_n$, a $2k+1$-window size moving average is a sequence of numbers $y_{k+1}, \ldots, y_{n-k}$, where y_i is the average of $x_{i-k}, \ldots, x_{i+k}$. Write a program which takes a sequence and the integer k as input, and prints out the $2k+1$ window-size moving average. Also plot the original sequence and the moving average on the graphics canvas.

12. The *Eratosthenes' Sieve* for determining whether a number n is prime is as follows. We first write down the numbers $2, \ldots, n$ on paper.[5] We then start with the first uncrossed number, and cross out all its proper multiples. Then we look for the next uncrossed number and cross out all its proper multiples, and so on. If n is not crossed out in this process, then it must be a prime. Write a program based on this idea.

13. Suppose we are given an array `marks` where `marks[i]` gives the marks of student with roll number `i`. We are required to print out the marks in non-increasing order, along with the roll number of the student who obtained the marks. Modify the sorting algorithm developed in the chapter to do this. *Hint*: Use an additional array `rollNo` such that `rollNo[i]` equals `i` initially. As you exchange marks during the course of the selection sort algorithm, move the roll number along with the marks.

14. Suppose you are given a sequence of numbers, preceded by the length of the sequence. You are required to sort them. In this exercise, you will do this using the so-called *Insertion sort* algorithm. The idea of the algorithm is to read the numbers into an array, but keep the array sorted as you read. In other words, after you read the first i numbers, you must make sure that they appear in the first i elements of the array in sorted (say non-increasing) order. So when you read the $i+1$th number, you must find where it should be inserted. Suppose you discover that it needs to be placed between the numbers that are currently at the jth and $j+1$th position, then you should move the numbers in positions $j+1$ through $i-1$ (note that the indices or positions start at 0) forward in the array by 1 step. Then the newly read number can be placed in the $j+1$th position. Write the program that does this.

15. A friend ("the magician") shows you a deck of cards. He has arranged it before hand in an interesting order because of which he can perform the following "trick". He picks up the top card, turns it face up, and it is seen to be the ace, which he puts away. He then takes the next card and puts it at the bottom of the deck without showing it to you. Then he shows you the card now at the top of the deck, which turns out to be the 2, which he also puts away. He repeats the following process until the deck has no cards. It turns out (magically!) that you see the cards in increasing face value, i.e. the first card to be exposed is the ace, then the 2, then the 3, then the 4, and so on,

[5] Well, Eratosthenes used a clay or wax tablet!

until the King. Of course, the "magic" is all in the order in which the cards were placed in the deck at the beginning. Write a program that explains the magic, i.e. figures out the initial order of the cards and prints it. *Hint*: Reverse the process.

16. Suppose we have received m applications for filling n jobs. For simplicity, assume that each applicant is interested in some 2 of the m jobs, and specifies these, in order of preference in his/her application. Suppose the application also contains the marks obtained by the applicant in an appropriate examination. The jobs are to be given to the applicants in decreasing order of their marks, i.e. first the applicant with the highest marks is to be given the job of his/her choice if still available, then the one with the next highest marks and so on. Write a program which makes this assignment.

17. Write a function which given polynomials $P(x), Q(x)$ returns their *composition* $R(x) = P(Q(x))$. Say $P(x) = x^2 + 3x + 5$ and $Q(x) = 3x^2 + 5x + 9$. Then $R(x) = (3x^2 + 5x + 9)^2 + 3(3x^2 + 5x + 9) + 5$.

18. Consider a long railway track divided into some n parts of possibly unequal lengths. For each ith part, you are given its length L_i and a maximum speed s_i with which trains can run on it. You are also given the data for a certain locomotive: its maximum speed s and the maximum accereration a it is capable of (assume this is independent of the speed, for simplicity), the maximum deceleration d (again independent of the speed) it is capable of. Suppose the train starts at rest at one end of the track and must come to rest at the other end. How quickly can the train complete this journey? Make sure your code works for all possible values of the parameters.

19. Write a program that takes in two numbers with 100 digits each, and prints out their product. Adapt the polynomial multiplication algorithm discussed in the text, or the way you did multiplication in primary school.

20. A permutation of a set S is simply a listing of the of the objects in a set. For example, if our set is $\{0, 1, 2\}$, then it has 6 permutations as follows.

012
021
102
120
201
210

Just as we have the notion of *lexicographic order* on character strings, we can have the notion of lexicographic order on permutations of a set. For this, we must first number the elements of the set from 0 to $n - 1$ (unless the elements are already such numbers). Suppose x, y are permutations of a set S. We scan the elements of x, y from the left until we find the first position at which the elements differ. If the element at that position in x has smaller number than the element at that position in y then we declare $x < y$. Otherwise we declare $y < x$.

Write a function that takes a permutation of the set $S = \{0, 1, \ldots, n - 1$ and returns the lexicographically next permutation if such a permutation exists. Thus, given the permutation 120 for the case $n = 3$, the program must return 201. *Hint*: Try out a few permutations to deduce the relationship between a permutation and the lexicographically next permutation.

21. Using the function developed above, write a program that prints out all permutations of a set $\{0, 1, \ldots, n - 1\}$, for arbitrary n.

CHAPTER 15

More on Arrays

We begin by considering the problem of representing textual data. In chapter 3 we discussed the `char` datatype for storing characters. However, we rarely work with single characters. More often, we will need to manipulate full words/sentences, or strings/sequences of characters. A character string is customarily represented in C as an array of characters. A better representation is available in C++, as we will see in Section 22.1. But it is worth knowing the array based representation because you will encounter it because of legacy reasons (e.g. Section 15.3.1) and because the C++ recommended representation builds upon this.

Next, we discuss *multidimensional arrays*. An ordinary (one-dimensional) array can be thought of as a sequence of values. A two-dimensional array can be thought of as a matrix or a table (rows and columns) of values. Two-dimensional arrays are very useful, especially in scientific computation. We will discuss an important use of two dimensional arrays: representing and solving linear systems of equations. C++ allows us to build our own representations for two dimensional arrays which have all features we discuss in this chapter, and some additional ones. This is discussed in Section 22.2.7.

So far we have been executing C++ programs by writing `a.out` or `./a.out`. However, it is possible to supply input to the program on the command line itself, e.g. by writing `a.out input-text`. This requires the use of character arrays, and is discussed in Section 15.3.1.

15.1 CHARACTER STRINGS

An array of characters can be defined just as you define arrays of `doubles` or `ints`.

```
char name[20], residence[50];
```

The above defines two arrays, `name` and `residence` of lengths 20 and 50, ostensibly for storing the name and the residence. Since we will usually not know the exact number of characters in a name or in an address, it is customary to define arrays of what we guess might be the largest possible length. This might seem wasteful, and it is, and we will see better alternatives in later chapters.

So if we want to store a character string "Shivaji" in the array, we will be storing 'S' in `name[0]`, 'h' in `name[1]`, and so on. The string is 7 characters long, and you would think that we should store this length somewhere. While printing the string for example, we clearly do not want the `name[7]` through `name[19]` printed. The convention used in the C language, and inherited into C++ from there, is that instead of storing the length explicitly, we store a special character at the end of the actual

string. The special character used is the one with ASCII value 0, and this can be written as '\0'. Note that '\0' is not printable, and is not expected to be a part of any real text string. So it unambiguously marks the end of the string.

Special constructs are provided for initializing character arrays. So indeed we may write

```
char name[20] = "Shivaji";
char residence[50] = "Main Palace, Raigad";
```

The character string "Shivaji" has 7 characters. So these will be placed in the first 7 elements of `name`. The eighth element, `name[7]`, will be set to `'\0'`. The character string "Main Palace, Raigad" has 19 characters. These and an additional '\0' will likewise be stored in the first 20 elements of the array `residence`. Thus, only the first 20 elements of `residence` will be initialized.

Here is an alternative form.

```
char name[] = "Shivaji";
char residence[] = "Main Palace, Raigad";
```

In this, C++ will calculate the lengths of `name` and `residence`. Following the previous discussion, these will be set to 8 and 20 respectively.

15.1.1 Output

Printing out the contents of a character array is simple. Assuming `name` is a character array as before,

```
cout << name;
```

would cause the contents of `name` from the beginning to the '\0' character to be printed on the screen.

The general form of the above statement is:

```
cout << charptr;
```

where `charptr` is an expression which evaluates to a pointer to a `char` type. If `name` is a character array, then `name` indeed is of type pointer to `char`. This statement causes characters starting from the address `charptr` to be printed, until a '\0' character is encountered. Thus character arrays passed to functions can be printed in the expected manner.

15.1.2 Input

To read in a string into a `char` array you may use the analogous form:

```
cin >> charptr;
```

Here, `charptr` could be the name of a `char` array, or more generally, an expression of type pointer to `char`. The statement will cause a whitespace delimited string typed by the user to be read into the memory starting at the address denoted by `charptr`. After storing the string, the '\0' character will be stored. Here is an example.

```
char name[20];
cout << "Please type your name: ";
cin >> name;
```

The second statement asks you to type your name and the third, `cin >> name;` reads in what you type into the array `name`. The following points are worth noting:

1. From what you type, the initial whitespace characters will be ignored. The character string starting with the first non-whitespace character and ending just before the following whitespace character will be taken and placed in `name`. Thus, if I type

   ```
   Abhiram Ranade
   ```

   with some leading whitespace, the leading whitespace will be dropped and only `"Abhiram"` would go into `name`. Next, following `"Abhiram"` a null character, i.e. '\0' would be stored. Thus the letters `'A'` through `'m'` would go into `name[0]` through `name[6]`, and `name[7]` would be set to '\0'.
2. This way of reading in text is not useful if the text contains spaces. Thus, in the above example, "Ranade" would *not* be read into `name`, even if I type it. For that it is necessary to use the `getline` command discussed below.
3. This statement is potentially unsafe. In the above example, if the user had typed in more than 20 characters without a whitespace in between, all those would be stored starting at `name[0]`. Thus, the characters read in would be stored past the end of the designated array `name`, possibly writing into memory that might have been allocated for some other variable.

The safe alternative to this is to use the following command.

```
cin.getline(x,n);
```

where `x` must be a name of a `char` array, or more generally a pointer to `char`, and `n` an integer. This will cause whatever the user types, including whitespace characters, to be placed starting from the address `x`, until one of the following occurs

- A newline character is typed[1] by the user. In this case, all characters up to the newline are copied into memory starting from the address `x`. The newline character is not copied. Instead, a '\0' is stored.
- `n-1` characters are typed without a newline. In this case, all the characters are placed into memory starting from address `x`, followed by a '\0' character.

As you may guess, it is customary to use the length of `x` as the argument `n`. So for example we can write

```
char name[20];
cin.getline(name,20);
```

In this case, at most 19 characters that the user types will be copied, and we will have no danger of overflowing past the array limit.

15.1.3 Character String Constant

Quoted text, such as `"Please type your name:"` constitutes a *string constant* in C++. The compiler stores the string somewhere in memory (followed by '\0'), and you may refer to it. Interestingly enough, the value of a string constant is not the text, but a pointer to the first character of the text. Thus, when you write

[1] On most modern keyboards this happens when you press the key labelled ENTER.

```
cout << "Please type your name:";
```

you are merely using the general form mentioned in Section 15.1.1. You may also write

```
char *name;
name = "Einstein";
```

Even in this statement, the right-hand side of the assignment, `"Einstein"` denotes the address in memory of where the text `"Einstein"` is stored (terminated by a '\0' as always). Thus it is fine to store this in a variable of type `char*`. Of course, if you subsequently write `cout << name;`, you will see `"Einstein"` printed.

The type of a character string constant is `const char*`.

15.1.4 Character-array Processing

Character arrays behave like ordinary integer arrays, except when it comes to reading and printing, and in that they contain a '\0' character which marks the end of the useful portion of the array. So processing them is reasonably straightforward. Note that characters are a subtype of integers, and as such we can perform arithmetic on characters, and compare them, just as we do for integers.

Our first example is a function for determining the length of the text stored in a char array.

```
int length(const char *txt){
  // precondition: txt points to sequence of '\0' terminated
  // characters.
  int L=0;
  while(txt[L] != '\0') L++;
  return L;
}
```

The function takes a single argument, say the array name (or the pointer to the zeroth element of the array). Notice that the actual length of the array is not needed. This is because we access elements only till the null character. Indeed, the function simply steps through the elements of the array, and returns the index at which it finds the null character, '\0'. Since the starting index is 0, the null character will be at index equal to the length of the text string. Note that we have marked `txt` with the keyword `const` because we don't expect to change it within the function.

Our second example is a function for copying a string stored in an array `source` to another array `destination`. This is like copying other arrays, except that we must only worry about the useful portion of the source array, i.e. till the occurrence of the '\0' character. The function does not worry at all about the lengths of the 2 arrays as defined, it is assumed that the call has been made ensuring that indices will not exceed the array bounds.

```
void scopy(char destination[], const char source[])
// precondition: '\0' must occur in source.  destination must be
// long
// enough to hold the entire source string + '\0'.
{
  int i;
  for(i=0; source[i] != '\0'; i++)
    destination[i]=source[i];
```

```
  destination[i]=source[i];    // copy the '\0' itself
}
```

As before, we have marked `source` constant because it does not change inside the function.

Here is a more interesting function: it takes two strings and returns which one is lexicographically smaller, i.e. would appear first in the dictionary. The function simply compares corresponding characters of the two strings, starting at the 0th. If the end of the strings is reached without finding unequal characters, then it means that the two strings are identical, in which case we must return `'='`. If at some comparison we find the character in one string to be smaller than the other, that string is declared smaller. If one string ends earlier, while the preceding characters are the same, then the string that ends is smaller.

This logic is implemented in the code below. We maintain the loop invariant: characters 0 through $i-1$ of both arrays must be non null and identical. So if we find both `a[i]` and `b[i]` to be null, clearly the strings are identical and hence we return 0. If `a[i]` is null but not `b[i]`, then `a` is a prefix of `b`. Because prefixes appear before longer strings in the dictionary, we return `'<'`. We proceed similarly if `b[i]` is null but not `a[i]`. If `a[i]>b[i]` we return `'>'`, if `a[i]<b[i]` we return `'<'`. If none of these conditions apply, then the `i`th character in both strings must be non-null and identical. So the invariant for the next iteration is satisfied. So we increment `i` and go to the next iteration.

```
char compare(char a[], char b[])
// returns '<' if a is smaller, '=' if equal, '>' if b is smaller.
{
  int i = 0;
  while(true){                  // Invariant: a[0..i-1] == b[0..i-1]
    if(a[i] == '\0' && b[i] == '\0') return '=';
    if(a[i] == '\0') return '<';
    if(b[i] == '\0') return '>';
    if(a[i]<b[i]) return '<';
    if(a[i]>b[i]) return '>';
    i++;
  }
}
```

This may be called using the following main program.

```
main(){
  char a[40], b[40];
  cin.getline(a,40);
  cin.getline(b,40);
  cout << a << " " << compare(a,b) << " " << b << endl;
}
```

If you execute this program, it would expect you to type two lines. Say you typed:

```
Mathematics
Biology
```

then it would print out > and stop, because "Mathematics" appears after "Biology" in the dictionary order.

15.1.5 Address Arithmetic

C++ programs processing `char` arrays often use arithmetic on addresses.

Suppose `x` is the name of an array of some type `T`. We have already said that `x` has value equal to the address of the zeroth element of the array. Suppose further that `i` is an integer expression. Then the expression

```
x+i
```

is valid in C++ programs, and has the value equal to the address of `x[i]`. Note that in general, a single element of type `T` may require some `s` bytes. Thus, while `x+i` seems to be adding `i` to the address `x`, the actual value added is `i*s` because the address of `x[0]` and of `x[i]` differ by `i*s` and not just `i`. Indeed, in general, if `x` is of type `T*`, then `x+i` is the address obtained by adding `i*s` to the address denoted by `x`. Since this can be somewhat confusing, we have not discussed such address arithmetic so far.

However, if `x` is of type `char*`, then `s` equals 1. In that case, when we write `x+i`, we indeed mean the address obtained by adding `i`. So perhaps for this reason, address arithmetic is quite common in character processing. Thus, the `scopy` function would more commonly be written as

```
void scopy(char *destination, const char *source){
  while(true){
    *destination = *source;
    if(*source == '\0') break;
    destination++;
    source++;
  }
}
```

15.2 TWO-DIMENSIONAL ARRAYS

Often we need to represent mathematical objects like matrices. For this, C++ provides two dimensional arrays. Here is an example of how a two dimensional array might be defined:

```
double a[m][n];
```

This causes space for `m*n` variables of type `double` to be allocated. These variables are accessed as `a[i][j]` where we require $0 \leq$ `i` $<$ `m`, and $0 \leq$ `j` $<$ `n`. The variables are stored in the so called row major order in memory, i.e. in the order `a[0][0], a[0][1], ... a[0][n-1], a[1][0], ... a[1][n-1], ... a[m-1][n-1]`. The numbers `m,n` are said to be the first and second dimension of the array. We will also refer to them as the number of rows and the number of columns respectively.

Manipulating two-dimensional arrays is similar to one-dimensional—we commonly use a loop to go over each dimension. As an example, consider the problem of multiplying two matrices. Remember that if A is an $m \times n$ matrix, and B an $n \times p$ matrix, then their product is an $m \times p$ matrix C where

$$c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj}$$

where we have let the array indices start at 1, as is customary in mathematics. The code below, of course, starts indices at 0. The code also shows how a two-dimensional array can be initialized in the definition itself if you wish. The values for each row must appear in braces, and these in turn in an outer pair of braces.

```
double a[3][2]={{1,2},{3,4},{5,6}}, b[2][4]={{1,2,3,4},{5,6,7,8}},
       c[3][4];

for(int i=0; i<3; i++)                          // compute c = a * b.
   for(int j=0; j<4; j++){
      c[i][j] = 0;
      for(int k=0; k<2; k++)
         c[i][j] += a[i][k]*b[k][j];
   }

for(int i=0; i<3; i++){                         // print out c.
   for(int j=0; j<4; j++) cout << c[i][j] << " ";
   cout << endl;
}
```

We may define two-dimensional arrays of `char`s, with initialization, which is of course optional. For example, we could write

```
char countries[6][20] = {"India","China","Sri Lanka","Nepal",
                         "Bangladesh","Pakistan"};
```

Here, the first string, `"India"` is deemed to initialize the zeroth row, and so on for the six strings.

Applying only one index to the name of a two-dimensional array returns the address of the zeroth element of the corresponding row. For character arrays, this is the way to refer to one of the strings stored. Thus, `countries[i]` will return the address of the zeroth character of the `i`th string stored in the array, in other words, the address of the `i`th string. So if we write `compare(countries[0], countries[1])`, where `compare` is as defined in Section 15.1.4, it would return `'<'` as the result because India will precede Sri Lanka in the dictionary order.

Here is a program which has two arrays, `countries` which lists countries, and `capitals` which lists corresponding capitals. It takes as input a string from the keyboard. It prints out the name of the corresponding capital if the string is in the list of countries stored in `countries`. This check is made using our `compare` function.

```
int main(){
  const int wordLength = 20;
  char countries[6][wordLength] = {"India","China","Sri Lanka",
                         "Nepal","Bangladesh","Pakistan"};
  char capitals[6][wordLength] = {"New Delhi","Beijing","Colombo",
                         "Kathmandu","Dhaka","Islamabad"};
  char country[wordLength];
  cout << "Country: ";
  cin.getline(country,wordLength);
```

```
  int i;
  for(i=0; i<6; i++){
    if(compare(country,countries[i]) == '='){
            // only one index applied to array countries
      cout << capitals[i] << endl;
      break;
    }
  }
  if(i == 6) cout << "Don't know the capital.\n";
}
```

When the loop terminates, we know that `i` must be strictly less than 6 if the country was found in `countries`, and equal to 6 if not found. Hence, we print the message that we don't know the capital only if `i` is 6 at the end.

15.2.1 Linear Simultaneous Equations

One of the most important uses of matrices and two dimensional arrays is to represent linear simultaneous equations. Say we are given simultaneous equations:

$$\begin{aligned} 3x_2 + 5x_3 &= 10 \\ 2x_1 + 6x_2 + 8x_3 &= 38 \\ 7x_1 + 4x_2 + 9x_3 &= 22 \end{aligned}$$

Then they can be conveniently represented by the matrix equation

$$\begin{bmatrix} 0 & 3 & 5 \\ 2 & 6 & 8 \\ 7 & 4 & 9 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 10 \\ 38 \\ 22 \end{bmatrix}$$

Denoting the matrix by A, the vector of unknowns by x and the right hand side vector by b, we have the matrix equation $Ax = b$ in which we are to solve for x given A, b.

The direct way to solve a system of equations is by a process called *Gaussian elimination*[2], in fact a form of it called Gauss–Jordan elimination.

Observe first that if the matrix A was the identity matrix, i.e. $a_{ii} = 1$ and $a_{ij} = 0$ for all $i, j \neq i$, then the problem is very easy. Multiplying out we would get $x = b$. Thus for this b is itself the solution. This suggests a strategy. We will make modifications to A, b such that the modifications do not change the solution of $Ax = b$. If at the end of the sequence of modifications, our matrix A becomes the identity matrix then the value of b at that time would itself be the solution.

It turns out that certain operations performed on the system of equations (and hence A, b) indeed do not change the solutions to the system. One such operation is to multiply any equation by a constant. This is akin to multiplying a row of the matrix A and the corresponding element of the vector b by a (the same) constant. Another operation is to add one equation to another, and replace the latter equation by the result. In our example, say we add the first equation to the second. Thus, we get the equation $2x_1 + 9x_2 + 13x_3 = 48$. We replace the second equation with this equation. This is succinctly done in the matrix representation: we merely add the first row of A to the second row, and the first element

[2] The method is actually much older than Gauss.

of b to the second element of b. Thus the second row of A would then become [2 9 13] and the second element of b would become 48, while the other elements remained the same.

We now show how we can change A, b, without changing the solution, so that the first column of A becomes 1,0,0 (read top to bottom), i.e. identical to the first column of the identity matrix. The same process can then be adapted for the other columns.

1. If the coefficient of x_1 is zero in the first equation, pick any equation which has a non zero coefficient for x_1. Suppose the ith equation has a non-zero coefficient for x_1. Then exchange equation 1 and equation i. This corresponds to exchanging row 1 and row i of A and also element 1 and element i of b. Doing this for our example, we get

$$\begin{bmatrix} 2 & 6 & 8 \\ 0 & 3 & 5 \\ 7 & 4 & 9 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 38 \\ 10 \\ 22 \end{bmatrix}$$

2. Divide the first equation by the coefficient of a_{11}. We thus get

$$\begin{bmatrix} 1 & 3 & 4 \\ 0 & 3 & 5 \\ 7 & 4 & 9 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 19 \\ 10 \\ 22 \end{bmatrix}$$

3. For each i, add $-a_{i1}$ times the first equation to equation i. Say we do this for row 2. Thus, we must add $-a_{21} = 0$ times the first row. So nothing need be done. So we then consider row 3. Since $a_{31} = 7$, we add -7 times the first equation to equation 3. Thus we now have:

$$\begin{bmatrix} 1 & 3 & 4 \\ 0 & 3 & 5 \\ 0 & -17 & -19 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 19 \\ 10 \\ -111 \end{bmatrix}$$

It should be clear that the above process would indeed make the first column identical to the first column of the identity matrix. In a similar manner, you should be able to get the other columns to match the identity matrix.

The first step in the above description deserves more explanation. Suppose you have managed to make the first $j-1$ columns of A resemble the first $j-1$ columns of the identity matrix. Now the first step above instructs you to find an equation in which the coefficient of x_j is non-zero. For this, you should only look at equations j through n, and not consider the first $j-1$ equations. This step may or may not succeed. It will not succeed if $a_{kj} = 0$ for all $k = j \ldots n$. In this case, it turns out that the system of equations does not have a unique solution; it may have many solutions or no solutions at all. In this case, you should report failure.

The code for doing all this is left as an exercise.

15.2.2 Passing Two-dimensional Arrays to Functions

It is possible to pass a two dimensional array to a function. However, in the called function, the second dimension of the array parameter must be given as a compile time constant. Thus, we might write

```
void print(char countries[][20], int noOfCountries){
  for(int i=0; i<noOfCountries; i++) cout << countries[i] << endl;
}
```

This may be called `print(countries,6)`, where the second argument is the first dimension of the `countries` array. It will print out the countries on separate lines.

This is not too useful, because any such function can only be used for arrays in which the second dimension is 20. For example, this makes it impossible to write a general matrix multiplication function for matrices of arbitrary sizes. This is a fundamental drawback of two dimensional arrays in the language C, which has been inherited into the language C++. In Section 22.2.7 we will see how it can be overcome quite elegantly using the flexible nature of C++.

But if we do know the second dimension, then the standard two dimensional arrays are useful. Here is how they can be used in drawing polygons in Simplecpp graphics.

15.2.3 Drawing Polygons in Simplecpp

Simplecpp contains the following command for drawing polygons:

```
Polygon pName(cx,cy,Vertices,n);
```

This will create a polygon named `pName`. The parameters `cx,cy` give the rotation center of the polygon. The parameter `n` is an integer giving the number of vertices, and `Vertices` is a two dimensional `double` array with `n` rows and 2 columns, where each row gives the x,y coordinates of the vertices, relative to the center `(cx,cy)`. A polygon is a shape in the sense of Chapter 5, so we may use all the commands for shapes on polygons.

The boundary of the polygon is traced starting at vertex 0, then going to vertex 1 and so on till vertex `n-1` and then back to vertex 0. Note that the boundary may intersect itself.

Here is an example. We create a regular pentagon and a pentagonal star. Then we rotate them.

```
int main(){
  initCanvas("Pentagon and Star");
  double pentaV[5][2], starV[5][2];

  for(int i=0; i<5; i++){
    pentaV[i][0] = 100 * cos(2*PI/5*i);
    pentaV[i][1] = 100 * sin(2*PI/5*i);
    starV[i][0] = 100 * cos(4*PI/5*i);
    starV[i][1] = 100 * sin(4*PI/5*i);
  }

  Polygon penta(200,200,pentaV,5);
  Polygon star(200,400,starV,5);

  for(int i=0; i<100; i++){
    penta.left(5);
    star.right(5);
    wait(0.1);
  }

  getClick();
}
```

Note that there is a more natural ways of specifying the star shape: consider it to be a (concave) polygon of 10 vertices. Thus, we could have given the coordinates of the 10 vertices in order. Calculating the coordinates of the "inner" vertices is a bit messy, though.

15.3 ARRAYS OF POINTERS

An array is really a sequence in memory of variables of the same type. We have seen arrays of `int`, `double`, `char`, but we can have arrays of any type of variable. So you might ask, can we have arrays of pointers? It is certainly possible, and it turns out to be useful too.

We can create an array of 10 variables, each of type pointer to `int` by writing the following.

```
int *y[10];
```

This statement is undoubtedly confusing. The way to understand it is to compare it with a usual array definition.

```
int x[10];
```

You can read this statement as saying "`x[i]` is an `int` for `i=0` to `i=9`." In a similar manner, you should read the statement `int *y[10];` as saying "`*(y[i])` is an `int` for `i=0` to `i=9`." But if content of `y[i]` is an `int`, then `y[i]` must be an `int` pointer.

Once you have defined an array of pointers, you can store addresses of appropriate variables in each element of the array. For example, you might write something like

```
int *y[10];
int z = 100;
y[0] = &z;
cout << *y[0] << endl;
```

This will print 100, because `y[0]` contains the address of `z`, and hence `*y[0]` just means `z`, and hence the value of `z`, 100, will be printed.

We next discuss an important use of arrays of pointers.

15.3.1 Command-line Arguments to `main`

So far, we have executed C++ programs by specifying the name of the executable file, usually `a.out`, on the command line. Specifically, the program is executed by typing `a.out` or `./a.out` on the shell command line. This causes the `main` function in your program to be called. But you may execute your program differently. C++ allows you to provide additional text after `a.out`, and this text can be processed by your program. For example, you may write

```
./a.out Mathematics Biology
```

In this case, your program can be told that you have typed the words `Mathematics` and `Biology` after `a.out`. This can be done using an alternative (overloaded) declaration provided for `main`.

```
int main(int argc, char *argv[]);
```

Thus, `main` may take two arguments. The first is an integer argument `argc`. The second argument is an array (since it ends in `[]`) has name `argv`, and each element is of type `char*`. In other words, `argv` is an array of pointers to `char`.

Suppose you use this form of `main`. Then when you execute your program, the Operating System calls the function `main`, but also passes some parameters. Specifically, the following are the values passed in the parameters:

1. The parameter `argc` gives the number of words typed on the command line, including the name of the executable program (`a.out` or other). Note that by "word" we simply mean white space delimited sequence of characters.
2. The parameter `argv` is an array of `argc` elements, with the `i`th element `argv[i]` being the address of the `i`th command line word (typically called `i`th command line argument).

Thus, if you had invoked the main program by writing `./a.out Mathematics Biology` the value of `argc` would be 3. The parameter `argv` would have 3 elements of type `char*`, and these would respectively be addresses of the text strings (null terminated) `"./a.out"`, `"Mathematics"`, and `"Biology"` respectively.

Here is a simple program that just prints out the values of all its command-line arguments.

```
int main(int argc, char *argv[]){
  for(int i=0; i<argc; i++) cout << argv[i] << endl;
}
```

This program when invoked as `a.out Mathematics Biology` would print out

```
a.out
Mathematics
Biology
```

Of course, you can do more interesting processing of the command line arguments. See Appendix E for an example.

15.4 MORE DIMENSIONS

Larger number of dimensions are also allowed. The statement below defines one 3 dimensional array and one 4 dimensional one.

```
int u[10][20][30], v[5][5][6][6];
```

These can be used in the natural manner.

15.5 CONCLUDING REMARKS

The two main topics of this chapter: using arrays to store strings and two dimensional arrays are legacies from the C language. We noted that in C++ you will more likely use the advanced constructs to be discussed in Sections 22.1 and 22.2.6. However, the basic notions such as how elements are accessed remain the same in the more advanced constructs.

Two dimensional arrays are very common in scientific computing. So it is useful to become familiar with using two indices to access the elements. It should be noted that accessing an element of a two-dimensional array also involves address calculation as was the case for one dimensional arrays. Of course, the calculation will involve both the indices. But even so, the time required to access an element will not depend upon how many rows or columns there are in the array.

The notation to represent pointer arrays is a bit cryptic, however you will want to become familiar with it because it is useful for accessing command-line arguments.

EXERCISES

1. Write a program that reads in an integer from the keyboard and prints it out in words. For example, on reading in 368, the program should print "three hundred and sixty-eight".
2. For this exercise it is important to know that the codes for the digits are consecutive, starting at 0. Further '8' − '0' is valid expression and evaluates to the difference in the code used to represent the characters, and is thus 8. To clarify, if we execute

   ```
   char text[10] = "1729";
   int d = text[1] - '0';
   ```

   Then `d` will have the value 7. Use this to write a function that takes a char array containing a number and return an integer of the corresponding magnitude.
3. Suppose `destination` and `source` are of type `char*`. What do you think the following statement does?

   ```
   while(*destination++ = *source++);
   ```

   Note: it uses several programming idioms you have been warned not to use. The point of this exercise is not to encourage the use of these idioms but to warn you how dense C++ code can be.
4. Extend the marks-display program of Section 14.2.2 to use names rather than roll numbers. At the beginning, the teacher enters the number of students. Then the program must prompt the teacher to enter the name of the student, followed by the marks. After all names and marks have been entered, the program then gets ready to answer student queries. Students enter their name and the program prints out the marks they have obtained.
5. Write a function which takes a sequence of parentheses, opening and closing, of all types, and says whether it is a valid parenthesization. In this exercise we use the term parenthesis to mean not just '(' and ')', but also '{', '[' and their matching counterparts, '}', ']'. Specifically, each opening parentheses should be matched to a closing parenthesis of the same type, occurring to the right of the opening parenthesis. Further, one matching pair must either be fully inside another matching pair or fully outside it.
6. Write a "calculator" program that takes three command-line arguments in addition to the name of the executable: the first and third being `double` values and the second being a single `char`. The second argument must be specified as an arithmetic operator, i.e. `+`, `-`, `*` or `/`. The program must perform the required operation on the two numbers and print the result. Appendix E will provide some useful ideas.

7. Write a program that solves a system of n linear equations in n unknowns, based on the discussion of Section 15.2.1. The number of unknowns, n, should be declared as a `const int`, e.g. as follows.

```
const int n = 4;
```

Then subsequently in your program you can define the arrays using `n` as defined above. This way it should be possible to use your program to solve systems of different size simply by changing the value of `n`.

8. Write a program that reads in a square matrix and prints its determinant. As above, make the dimension of the matrix a `const int`. You should NOT use the recursive definition of determinant, but instead use the following properties:

- Adding a multiple of one row to another leaves the determinant unchanged.
- Exchanging a pair of rows causes the determinant to be multiplied by -1.
- The deteminant of an upper triangular matrix (all zeros below the diagonal) is simply the product of the elements on the diagonal.

9. Cramer's rule states that the solution to a system of linear equations $Ax = b$ is given as $x_i = \frac{\text{Determinant}(A_i)}{\text{Determinant}(A)}$ where A_i denotes the matrix obtained by replacing the ith column of A by b. Use the code developed in the previous exercise to find the solution to $Ax = b$. Note by the way that this method is slower than the Gaussian elimination algorithm given in the text.

10. Suppose an $n \times n$ array `A` is used to hold a black-and-white image. Specifically, assume that `A` only contains 0s and 1s. Two array elements are considered adjacent if one of their indices is the same and the other differs exactly by 1. Thus, `A[i][j], A[i+1][j]` are adjacent, and also `A[i][j], A[i][j+1]`. Further, `A[i][j] = 1` denotes the presence of an "object", and `A[i][j] = 0` the absence of an object. If a pair of adjacent array elements are both 1, then they are said to be a part of the same object. This rule can be applied repeatedly, e.g. if an entire row is 1, then it is a part of a single object. Indeed, two elements are considered to be a part of the same object only if this is implied by repeated application of the rule.

Write a program that counts the number of different objects in the image. You may find it convenient to modify the array. Make sure that your program will count correctly no matter what shape each object has. This is a tricky problem.

CHAPTER 16

Arrays and Recursion

Many problems on sets/sequences have elegant solutions using recursion. Suppose a problem is given to you involving a sequence of length n. You can solve it using recursion if the following hold.

1. If n is small, say $n = 1$, then you have a way of solving the problem.
2. If n is not small, say $n > 1$, you have a way to construct smaller problem(s) of the same kind, on smaller sequence(s) such that from the solution to the smaller problem(s) you will be able to construct a solution to the original problem.

If you can do these steps, then you have a recursive algorithm, with the set/sequence typically stored in an array. Often, such recursive algorithms are very simple to state and code, and also fast. In what follows, we will see examples of this approach.

16.1 BINARY SEARCH

We often sort data because it looks nice to print it that way. However, there is another important motivation. Certain operations can be performed very fast if the data is sorted.

Suppose we have an array in which we have stored numbers. Suppose we are subsequently given a number x and we are to determine if x is is present in the array. The natural strategy is to go over each element in the array and check if it equals x. In the worst case we might have to examine every array element.

We can adopt a cleverer strategy if the array is sorted. Say our array is `A` and it contains `size` elements. Say `A` is sorted in non-decreasing order. The basic idea is: instead of examining elements from the beginning of the array, in the first step we examine the element that is roughly in the middle of the array. Thus, in the first step we check if `x < A[size/2]`. Note that `size/2` means the value of `size` divided by 2 and rounded down. There are 2 cases to consider.

The check succeeds, i.e. `x` is smaller than `A[size/2]`. Now because the array is sorted, we know that all elements in the subarray `A[size/2..size-1]` will all be larger than `x`. Hence `x`, if present in the array, will be in the portion `A[0..size/2-1]`. Thus, using just 1 comparison, we have narrowed our search to the first half of the array.

The check fails, i.e. `x` is greater or equal to `A[size/2]`. Then we know that if `x` is present in `A` it must certainly be present in `A[size/2..size-1]`. Thus, it suffices to search subsequently only in `A[size/2..size-1]`.

Thus, in both cases, after one comparison, we have ensured that subsequently we only need to search in one of the halves of the array. But we can recurse on the halves!

The key question is: when does the recursion end. Clearly, if our array has only one element, then we should not try to halve it! In this case we merely check if the element equals `x` and return the result of the comparison.

This gives us the following recursive function.

```
bool Bsearch(int x, int A[], int start, int size){
// x : target value to search
// range to search: A[start..start+size-1]
// precondition: size > 0;
//
  if(size == 1) return (A[start] == x);
  int half = size/2;          // 0 < half < size, because size>1.
  if(x < A[start+half])
    return Bsearch(x, A, start, half); // recurse on first half
  else
    return Bsearch(x, A, start+half, size-half);
                                      // recurse on second half.
}
```

There is an extra parameter, `start` which says where the subarray starts. So we are searching in the region `A[start...start+size-1]`. The "middle" element now is `A[start+size/2]` which is the same as `A[start+half]` in the code. The "first half" starts at `A[start]` and has size equal to `half`. The "second half" starts at `A[start+half]` and has size `size - half`. Thus we have the recursive calls in the function.

Our code might look "obviously correct", but this is deceptive. Folklore has it that even experienced programmers make mistakes while writing binary search. So it is a good idea to check that our function indeed works correctly.

There are two aspects to working correctly: the function must terminate, and on termination return the correct answer. We first check that the function will indeed terminate. Clearly, when `size` becomes 1, the function will return. But note that if `size > 1`, the value `half = size/2` (integer division) is strictly between 0 and `size`. Thus we can conclude that `half` as well as `size - half` are both smaller than `size`. Hence, we have established that the second parameter to `Bsearch` always reduces, and hence must eventually become 1, where upon the function will return. That the correct value is returned follows in the manner we have argued above.

Here is a main program which tests our function.

```
int main(){
  const int size=10;
  int A[size]={-1, 2, 2, 3, 10, 15, 15, 25, 28, 30};
  for(int i=0; i<size; i++) cout << A[i] << " ";
  cout << endl;
```

```
  for(int x=-10; x<=40; x++)
    cout << x << ": " << Bsearch(x, A, 0, size) <<endl;
}
```

We search the array for the presence of every integer between -10 and 40. You will see that 1 is returned only for those integers that are present.

Notice that the array is sorted, but contains repeated values.

16.1.1 Estimate of Time Taken

Let us analyze a bigger example. Suppose we are checking for the presence of a number in an array of size 1024. How many array elements do we compare in the process?

The function `binsearch` will first be called with the `size` parameter equal to 1024. When we recurse, no matter how the comparison comes out, we will next call `binsearch` with `size` 512. Subsequently we call `binsearch` with `size` 256 and so on. Thus a total of 10 calls will be made: in the last call `size` will become 1 and we will return the answer. In each call we make only one comparison `x < A[start+half]`, and hence only 10 comparisons will be made!

Compare this with the case in which the array is not sorted: then we might have to make as many as 1024 comparisons! Even if we agree that it takes a bit longer to call a function, calling `binsearch` 10 times (including the recursion) will be much faster than having to possibly compare `x` with each of the 1024 elements.[1] Actually, our binary search can be written out as a loop, without recursion, the exercises ask you to do this.

In general, you can see that for the recursive algorithm the number of comparisons made is simply the number of times you have to divide the size of the array so as to get the number 1. This number is $\log_2 n$, if n denotes the size of the array. In other words, the time is proportional to $\log_2 n$ when we do Binary search.

In contrast, if the array is not sorted, we are forced to do linear search, in which case the we may need to compare `x` to all the elements in the array, i.e. there could be as many as n comparisons.

Binary search is a simple but important idea. You will see that it will appear in many places, perhaps slightly disguised, as it did in the *Bisection algorithm* (Section 8.3) for finding roots.

16.2 MERGE SORT

In Section 14.5, we saw the selection sort algorithm for sorting an array. In the worst case, selection sort will take time proportional to n^2, where n denotes the number of elements, often called *keys*, that are to be sorted. In this section, we will see the *Merge sort* algorithm which will take time proportional to $n \log_2 n$. As you can see, $\log_2 n$ is much smaller than n, and hence $n \log_2 n$ is much smaller than n^2. Indeed if you code up the two algorithms you will see that Merge sort runs much faster.

Merge sort is a recursive algorithm. If we want to sort the sequence `S`, we divide it into two sequences `U` and `V` of roughly equal size. We sort `U,V`, and then combine the results to get a single sorted sequence. This is our final result, the result of sorting `S`. Such an algorithm is also often called

[1] If `x` is not present in the array, we will know that only after comparing it with all the 1024 elements. If `x` is present, we will stop after we find it. So in this case, you could say that "on the average" we will compare `x` with half the elements, i.e. we will do 512 comparisons.

a *divide-and-conquer* algorithm, because we divide `S` into smaller sequences which we sort (conquer!) separately.

The division of `S` into `U,V` is simple: we just put the first half of `S` into `U` and the second half into `V`. The key question, of course, is how to combine, or *merge*, the results of sorting `U` and `V`. We will discuss this first, and then discuss the entire algorithm.

We should note that C++ already provides you with a function for sorting. This is discussed in Section 22.3.2. This function will also run in time proportional to $n \log_2 n$.

16.2.1 A Merging Algorithm

Suppose we are given two rows of students, in each of which the students are arranged in non-decreasing order of their height. Can we put them into a single row such in which the students will still be arranged in non-decreasing order? This problem is perhaps easier to visualize, but as you can see it is the same problem as that of merging two sorted sequences.

Here is a procedure to merge the two rows u, v of students into a single row s. The first student in s should be the shortest of all students. The shortest in u is the student at the front of u, and the shortest in v the student at the front of v. Thus, the shortest overall must be the shorter of the students at the front of u and front of v. So we can ask the shorter of the two to leave his/her row, and join row s. Next, we have to find the second shortest student. Since the shortest student has moved to s already, the second shortest must be the shortest from those that remain. So we again pick the shorter of the students at the front of the rows u, v, and send that student to the *back* of row s. For the third shortest, we merely repeat the procedure! Eventually, it might so happen that all the students in one of the rows u, v have left for s. Once this happens, we ask the students from the remaining row to join s, in the order they are standing in their row.

The analogy to the sorted sequences `U,V` should be clear. In fact, we will think of each of the sequences `S,U,V` as a queue, like the queue of drivers we had for the taxi dispatch problem (Section 14.2.6). Drivers were joining that queue at the end, just as students/keys will join `S` at the end. Drivers left from the front of the queue, and similarly in this case students/keys will leave `U,V` from the front. Thus the algorithm can be coded up as shown in Figure 16.1. The comments in the code explain the algorithm fully. As you can see, it matches the student row merging procedure discussed above. Also, you should be able to prove the correctness of the invariant given.

The function `Merge` executes `uLength + vLength` iterations, i.e. as many as the total number of keys. In each iteration a fixed number of instructions is executed. Hence we can say that the total time is at most some constant times the number of keys, i.e. proportional to the total number of keys.

16.2.2 Mergesort Algorithm

Given a merge algorithm, the mergesort algorithm is easy. For sorting a sequence `S` of length `n`, we proceed as follows.

1. Create two smaller arrays of roughly half the size. Say array `U` of size `n/2`, and array `V` of size `n - n/2`.
2. Copy `n/2` elements of `S` to `U` and the remaining `n-n/2` elements to `V`.
3. Get the arrays `U` and `V` sorted. This requires a recursive call for each.

```
void merge(int U[], int uLength, int V[], int vLength, int S[]){
// arrays U,V of length uLength and vLength respectively contain the
// sequences that are sorted in non-decreasing order.  The result of
// merging is to be placed in the array S.  The length of S is not
// specified explicitly, but it is assumed (precondition) to be
// uLength + vLength.

  for(int uFront=0, vFront=0, sBack=0; sBack<uLength+vLength;
  sBack++){
  // INVARIANT: sBack = uFront + vFront.  Keys U[0..uFront-1] will
  // have been moved to S, and and also keys V[0..vFront-1].  S will
  // contain these keys in S[0..sBack-1], in non-decreasing order.

    if(uFront<uLength && vFront<vLength){// if both queues non-empty
      if(U[uFront] < V[vFront]){   // if U has smaller
        S[sBack] = U[uFront];      // move to S
        uFront++;                  // advance U
      }
      else{                        // if V has smaller
        S[sBack] = V[vFront];      // move to S
        vFront++;                  // advance V
      }
    }
    else if(uFront < uLength) {    // else if only U is not empty
      S[sBack] = U[uFront];        // move to S
      uFront++;                    // advance U
    }
    else {                         // else if only V is not empty
      S[sBack] = V[vFront];        // move to S
      vFront++;                    // advance V
    }
  }
}
```

Fig. 16.1 *Merging algorithm*

4. Merge the arrays `U` and `V` back and put the result into the array `S`.

The code for `mergesort` follows this outline exactly.

```
void mergesort(int S[], int n){
  if(n>1){
    int U[n/2], V[n-n/2];
    for(int i=0; i<n/2; i++) U[i] = S[i];
    for(int i=n/2; i<n; i++) V[i-n/2] = S[i];

    mergesort(U, n/2);
```

```
    mergesort(V, n - n/2);
    merge(U, n/2, V, n-n/2, S);
  }
}
```

Note that we wrote the number of elements to be copied to `U` as `n - n/2` and not `n/2` in order to account for the possibility that `n` might be odd.

16.2.3 Time Analysis

We will now estimate the time $T(n)$ taken by `mergesort` to sort a sequence of length n. Initially, we copy the elements of `S` to `U,V`. As discussed above, this takes total time proportional to at most n. After that we call `mergesort` recursively. This takes time $T(n/2)$ and $T(n - n/2)$. Finally, we call `merge`. The time taken by merge, we said is at most proportional to n the total number of keys. Thus, we have

$$T(n) \leq \text{(Time proportional to } n) + T(n/2) + T(n - n/2)$$

where we have clubbed together the time to copy and time to merge as a single entry, proportional to n. Let c denote the constant of proportionality. Further, to simplify the algebra, let us assume that n is even. Then we have

$$T(n) \leq cn + 2T(n/2) \tag{16.1}$$

Note that if $m = n/2$ is also even, then our inequality above will also apply to m, i.e. we will have $T(m) \leq cm + 2T(m/2)$, or in other words

$$T(n/2) \leq cn/2 + 2T(n/4) \tag{16.2}$$

Here, we need $n/2$ to be even, or n to be a multiple of 4. But now we can substitute Eq. (16.2) into Eq. (16.1) and get

$$T(n) \leq cn + 2T(n/2) \leq cn + 2(c(n/2) + 2T(n/4)) = 2cn + 4T(n/4)$$

Suppose now that $n = 2^k$. Then we can continue the above process for k steps. Thus we will get

$$T(n) \leq kcn + 2^k T(n/2^k)$$

Since $n/2^k = 1$ and $k = \log_2 n$, we have

$$T(n) \leq cn \log_2 n + nT(1)$$

Noting that and $T(1) \leq c'$ for some c', we get $T(n) \leq cn \log_2 n + c'n \leq c''n \log_2 n$ for some constant c''. Thus we have shown that $T(n)$ is at most proportional to $n \log_2 n$.

Mergesort is a classical algorithm, and the idea of dividing the input into equal sized parts and recursing on the parts works in many other problems too.

16.3 THE 8 QUEENS PROBLEM

In this section, we will write a program to solve a well known puzzle: how do you place 8 queens on a chess board so that they do not capture each other? As you may know, a chessboard consists of 64 squares arranged in 8 rows and 8 columns. Each *chess piece* such as a queen must be placed on a unique square on the board. Two queens can capture each other if they are in the same row or the same

column, or if they are in the same major or minor diagonal, i.e. if the queens are in a line oriented at 45° to the rows.

Our program is somewhat clever. It will effectively try out all possible ways of placing queens on the chessboard and check if in any one of them no queen captures another. The key ingredient in it is the process by which we will explore all ways of placing queens. Many problems can be solved using the idea of "trying all possibilities". Thus, the idea behind the 8 queens program will work for many other problems, and indeed that is the reason we are studying this puzzle.

First, we need a way of representing the placement of queens on a chessboard. The most natural representation is to have a 8×8 two dimensional array of `bool`s. Element `[i][j]` of the array will be set `true` if and only if a queen is placed in the square in the `i`th row and `j`th column. This is an acceptable representation, but there is a better one available. Since we don't want the queens to capture each other, we know that we must have exactly one queen in each column: we need not even consider configurations in which two queens are in the same column. Thus, we can have an array `Q` of 8 integers, of which the `i`th element `Q[i]` would denote the row position of the queen in column `i`. Thus, our goal would be to fill this array with suitable numbers such that the queen positions that the numbers represent are non-capturing.

So the next question is: given a set of queen positions, how do we determine whether any pair of queens capture each other? More specifically, we know that the queen in column `j` is in row `Q[j]`, and that the queen in column `k` is in row `Q[k]`. Can we say whether or not they capture each other? For this, we need to convert the English-language description of what it means for queens to capture each other into numerical conditions. As discussed above, two queens capture each other if one of the following conditions hold.

1. The two queens are in the same row. Being in the same row simply means whether `Q[j]==Q[k]`. So we must check this.
2. The two queens are in the same column. The way we are representing queens, it is impossible to have two queens in the same column. Thus this check does not need to be made.
3. The two queens are in the same major or minor diagonal, or are in a line at $45\circ$ to the rows. This condition is satisfied if horizontal distance between the queens, `abs(j-k)`, is the same as the vertical distance `abs(Q[j]-Q[k])`. Thus, we must test this in our code.

Thus, we already have one ingredient of our algorithm. Suppose we somehow have our array `Q` filled with queen positions. Then the following function would determine whether any pair of queens captures each other.

```
bool capture(int Q[], int n)
// Returns true if queens stored in Q[0..n-1] capture each other.
{
  for(int j=1; j<n; j++){              // check for all pairs j != k
    for(int k=0; k<j; k++){
      if((Q[j] == Q[k]) ||               // in same row
         (abs(j-k) == abs(Q[j] - Q[k]))) // in same diagonal
        return true;
    }
  }
```

```
  return false;
}
```

The code considers an `n` by `n` board for the sake of generality. It checks for all distinct pairs `j, k` whether the queen in column `j` captures the queen in column `k`. If any capturing pair is found, `true` is returned immediately. If no capturing pair is found, even after considering all possible pairs, the algorithm returns `false`.

Now we turn to the harder problem of generating all possible queen configurations. As mentioned above, we will some how generate all possible queen configurations (one queen in each column), and call `capture` on each configuration. If `capture` returns `false` for any configuration, we can print out that configuration.

The key idea is to observe that the set S of all possible queen configurations has the following structure. This set must contain configurations in which the zeroth queen (i.e. the queen in column 0) is in row 0. It must also contain configurations in which the zeroth queen is in row 1, and so on. In other words, if we denote by S_i the set of configurations in which the zeroth queen is in row i, then we have

$$S = S_0 \cup S_1 \cup \ldots \cup S_{n-1}$$

Clearly, if we know how to generate each S_i, and if we do so for $i = 0, \ldots, n-1$, then we will have generated all of S.

But do we know how to generate S_i? Interestingly enough, S_i can also be thought of as a union of sets of configurations just like S. We will consider this in general. Suppose $S_{i_0,\ldots,i_{k-1}}$ denotes the set of configurations in which the queen in column 0 is in row i_0, the queen in column 1 in row i_1, and so on till the queen in column $k-1$ is in row i_{k-1}. Then if $k < n$, this set is also a union:

$$S_{i_0,\ldots,i_{k-1}} = S_{i_0,\ldots,i_{k-1}0} \cup S_{i_0,\ldots,i_{k-1}1} \cup \ldots \cup S_{i_0,\ldots,i_{k-1}n} \tag{16.3}$$

Thus, if we want to explore $S_{i_0,\ldots,i_{k-1}}$, we should just explore $S_{i_0,\ldots,i_{k-1}j}$ for $j = 0, \ldots, n-1$.

Thus we should consider a recursive algorithm. Equation (16.3) tells us how to recurse, we just need the base case. But that is easy. If $k = n$, then the set $S_{i_0,\ldots,i_{k-1}} = S_{i_0,\ldots,i_{n-1}}$ is simply the set of configurations in which all queens have been assigned a row position. Thus, it contains just one configuration: the one in which the queen in column j is in row i_j for $j = 0, \ldots, n-1$. Thus when $k = n$, we have placed all the queens, and so we should check if any pair captures each other. So we can call `capture`, and if it returns `false`, then we can print the configuration.

```
void search(int Q[], int k, int n){
// Search the set of queen configurations in which first k
// queens are in
// rows Q[0],...,Q[k-1].
// Of these print those configurations in which queens do not
// capture each other.
// n = total number of queens.
  if(k == n){
    if (!capture(Q, k)){
      for(int j=0; j<k; j++) cout << Q[j];
      cout << endl;
    }
  }
```

```
  else{
    for(int j=0; j<n; j++){
      Q[k] = j;
      search(Q, k+1, n);
    }
  }
}
```

There is a subtle point to be noted. When we call `search(Q, k+1, n)`, we must make sure that the the subarray `Q[0..k-1]` contains the same values as were there at the time of the original call. This is indeed true, because as you can see, a call `search(Q, k+1, n)` does not modify `Q[0..k]`.

The main program for calling this is very simple.

```
int main(){
  const int n=8;
  int Q[n];
  search(Q,0,n);
}
```

16.3.1 Enforcing Constraints Early

It is possible to speed up the program given above. In the jth iteration, we search through the set $S_{i_0,\ldots,i_{k-1}j}$. This is simply the set of positions in which we have placed the queens in columns 0 through $k-1$ in rows i_0 through i_{k-1}, and the queen in column k in row j.

Suppose now that the queen in column k when placed in row j for some j captures some queen in the previous columns. In that case, we already know that there can be no non capturing positions in set $S_{i_0,\ldots,i_{k-1}j}$. So why even bother to recursively call the search function for this set?

Not having to make a recursive call is a significant amount of saving. So indeed, after placing a queen in column k, we will check if it is captured by any of the previously placed queens. We will make the recursive call only if the new queen is not captured. But in this case, when `search` finally gets called with $i = n$, we know that all the queens that we have placed must not be capturing each other. So we can immediately print the position out.

Thus, we get the improved code of Figure 16.2. Notice that the function `lastCaptures` in this code is the analogue of the function `capture`. The new function merely checks whether the last placed (`k`th) queen captures the previous queens.

16.4 CONCLUDING REMARKS

Recursion is one of the most powerful strategies for designing algorithms. There are various ways in which recursion can work with sets/sequences: we can divide the sequence into two and may need to solve our problem recursively on both halves (merge sort) or only one half (binary search). It is also possible that we may view a sequence of length n as a single element followed by a sequence of length $n-1$. This view will require us to recurse on the length $n-1$ subsequence. So long as we can handle the base cases (sequences of some fixed length) directly, any such sequence splitting strategy will give us a recursive algorithm, provided we can build a solution for the larger sequence from the solution for the smaller sequence(s). Recursion is used for many, many problems. Here is a classic problem which is solved using recursion: given the coordinates of a set of points in the plane, find the closest pair, i.e.

```
bool lastCaptures(int Q[], int k)
// checks whether the queen in column k captures those in
// columns 0..k-1
{
  for(int j=0; j<k; j++){
    if((Q[j] == Q[k]) ||                  // in same row
       (abs(j-k) == abs(Q[j] - Q[k])))  // in same diagonal
      return true;
  }
  return false;
}

void search(int Q[], int k, int n){
  if(k == n){
    for(int j=0; j<k; j++)
      cout << Q[j];
    cout << endl;
  }
  else{
    for(int j=0; j<n; j++){
      Q[k] = j;
      if(!lastCaptures(Q, k)) search(Q, k+1, n);
    }
  }
}

int main(){
  const int n=8;
  int Q[n];
  search(Q,0,n);
}
```

Fig. 16.2 *Final n queens program*

a pair of points which are the smallest distance apart. The algorithm for this is somewhat involved, and will not be covered in this book.

An important point to remember while designing recursive algorithms: if the size of the sequence is n, and we wish to divide it into two parts, it is tempting to write the sizes of the parts as both $n/2$. However, note that n can be odd, and hence the sizes are better written as $n/2, n - n/2$.

The n queens problem is an example of the general constraint satisfaction paradigm. Problems in this paradigm have the following general form: find values for a set of variables such that the values satisfy some specified constraints. In the n queens problem, the variables were the positions `Q[0]` through `Q[n-1]` of the queens, and these variables had to satisfy the constraint that the queens when placed at positions given by the variables would not capture each other. We showed how the constraints can be expressed arithmetically, i.e. `Q[j] != Q[k]`, and `abs(j-k) != abs(Q[j]-Q[k])`. The solution idea was: systematically go through all possible choices of values for the variables, and

for each choice check if the constraints are satisfied. This strategy of solving is called *backtracking*, because we try out one value for a variable, and if that does not work, we return (backtrack) and try another. The idea of enforcing constraints early can speed up backtracking.

The constraint satisfaction paradigm appears often, as you will see in the exercises. Backtracking is a simple general strategy for solving constraint satisfaction problems.

EXERCISES

1. The binary-search algorithm can also be written using a loop, rather than using recursion. Do so. State the appropriate loop invariant and the potential function and argue correctness.
2. Design an input instance for the `mergesort` algorithm such that every line of code in the `merge` algorithm of Section 16.2.1 will execute in one of the calls to `merge`.
3. Suppose you are given two sorted sequences S, T of lengths m, n. Write a program that finds the median of their union. You may find it easier to write a program that finds the ith smallest in the union, for general i. *Hint*: Compare the medians of the sequences S, T. What does the comparison tell you about the position of the ith smallest?
4. A very popular and elegant algorithm for sorting is the so called *Quicksort*. If A is the sequence to be sorted, this works as follows.
   1. Pick a random element r of A. This element is often called a *splitter*.
   2. Construct a sequence S consisting of all elements smaller than r.
   3. Construct a sequence L consisting of the remaining elements.
   4. Sort the sequnces S, L (recursively!) to produce sequences S', L'.
   5. Return the concatenation of sequences S', L'.

   Write the program for Quicksort. By and large, Quicksort works very fast. More precisely, it is possible to show that the expected time taken by Quicksort (expectation calculated over all random choices of r in all calls) is proportional $n \log_2 n$. The proof of this is outside the scope of this book.
5. An interesting trick is employed to make Quicksort run fast. If the original sequence A is stored in the array `A`, then it is possible to ensure that steps 2,3 above will construct S, L in `A` itself, with S preceding L. This will ensure that the sorting step will also produce the result *in-place*, i.e. S', L' will be produced in the same (sub)arrays as were occupied by S, L. Thus, the last step, concatenation, does not have to be done explicitly. Here is how we can create S, L inside `A` itself. Start scanning from `A[0]` towards higher indices. Stop when you find a number `A[i]` larger than or equal to the splitter r. Now start scanning backwards from the end, `A[n-1]`. Stop when you find `A[j]` smaller than r. Exchange the elements `A[i],A[j]`. Clearly, `A[0..i]` and `A[j..n-1]` can be considered parts of `S,L`. We can extend these by repeating the process on the sub-array `A[i+1..j-1]`.
   Code up this idea. Write clear invariants to guide your code. There is great potential here for making silly mistakes!
6. How do you represent a curve on a computer, e.g. the curves used for drawing the letter $\mathcal{S}$? One possibility is to approximate it by a sequence of straight-line segments and circular arcs. A more general representation uses the so called *Bezier curves*, named after Pierre Bezier who invented them for representing automobile body designs. A Bezier curve of order n is defined by n *control*

points $p_1, \ldots, p_n$. The curve begins at p_1 and ends at p_n and is smooth. The other control points "attract" the curve towards them, but the curve need not pass through them. The curve consists of the path traced by $B_{p_1,\ldots,p_n}(t)$ as t goes from 0 to 1. Here, $B_{p_1,\ldots,p_n}$ is a function which maps real numbers to points on the plane, and is defined recursively as follows.

$$B_{p_1,\ldots,p_n}(t) = B_{q_1,\ldots,q_{n-1}}(t)$$

where

$$q_i = tp_i + (1-t)p_{i+1}$$

i.e. q_i is the point dividing the line segment p_ip_{i+1} in the ratio $t : 1-t$. The base case is

$$B_p(t) = p$$

Write a program which receives points $p_1, \ldots, p_n$ on the graphics canvas and plots the Bezier curve defined by them. Vary t in the interval [0,1] in small steps, say $\Delta = 0.01$, and join $B_{p_1,\ldots,p_n}(t)$ to $B_{p_1,\ldots,p_n}(t+\Delta)$ to get the curve. Experiment for different values of n and positions of p_i.

7. Modify the n queens code so that it returns as soon as one non-capturing configuration is found. The array argument should hold this configuration on return. If no non-capturing configuration is found the function should return `false`.

8. Modify the n queens codes so that it counts how many times `capture` and `lastCaptures` are called. These numbers will tell you whether enforcement constraints early helps. *Hint*: initialize a counter variable to 0 in the main program, and pass it by reference as needed.

9. If you substitute a unique digit for each of the letters in the following "sum", it becomes a valid arithmetic addition, as you might have written in primary school. Write a program which determines what to substitute for each letter. As you can see, this is a constraint satisfaction problem which can be solved by backtracking.

		S	E	N	D
+		M	O	R	E
	M	O	N	E	Y

10. Consider Exercise 16 of Chapter 14. Suppose that you now decide to ignore the marks obtained by the candidates. Further your goal now is to fill each job with a candidate who has indicated a preference for it (first or second, does not matter). Write a program which reads in an additional input, t, and determine whether at least t applicants can be assigned jobs. Solve this using backtracking. There exist faster algorithms than backtracking, but they are technically more involved and hence outside the scope of this book.

11. In the knapsack Problem, the input consists of numbers v_i, w_i for $i = 0$ to $n-1$, and a number C. The numbers v_i, w_i denote the monetary value and weight respectively of the ith object in a collection, and C the weight-carrying capacity of a knapsack. The goal is to pack the knapsack with objects of maximum total value possible, while not exceeding the weight capacity.
Solve this using backtracking.

12. Try to make your backtrack program for knapsack as efficient as possible. A simple idea is: don't wait to check the capacity condition until the entire candidate solution is constructed. If the first few items selected already exceed the knapsack capacity, then there is no need to consider all possible ways of selecting the subsequent items.

CHAPTER 17

Object-oriented Programming: Structures

How does one write a large program, or in general execute any large project? How do you begin, in what order do you do the things that need to be done? Sometimes, it might seem that there is a natural order: if you are constructing a building, perhaps you should build the foundation first, and then work your way up.[1] But this is too simplistic; clearly when you build the foundation you must already have designed the entire building. So that raises the question: in what order do you do the design? Such questions are not easy to answer also in the context of programming. But good answers can certainly be useful.

In this chapter and the next, we give an introduction to the leading methodology for program design, the so called *Object-Oriented Programming* (OOP) *methodology*. Here is a very rough overview of OOP. The first step is of course to clearly understand the specifications. What are the program inputs and what is expected as output? Once this is done, the next step, distinctive of OOP, is to focus on the entities involved in the program. As an example, consider a program that keeps track of the books in a library. In this, the entities presumably are the books and the patrons. If, for example, your program is concerned with predicting the movement of stars in a galaxy, the entities are clearly the stars and other heavenly bodies. Each entity is associated with some information, fixed and variable. For example, a book has a title and an author which do not change. But for each book you also need to keep track of whether it has been borrowed, and if so by whom. This information which can change constitutes its *state*. Likewise stars will have a mass, which likely does not change. But a star also has state, for example its position and velocity. The entities may change state spontaneously over time, or because of interaction with other entities. As per OOP methodology, it is crucial to understand the entities, their state and their interaction.

Once the entities, the state, and the interactions are understood, OOP requires that organization of the code mirror the entities and their relationships. More specifically, the data and the code required by the entire program should be partitioned amongst the different entities. Thus, our code will have as many parts as there are entities. Once we identify the parts, we work on each methodically: we define

[1] It is believed that the Kailashnath temple in the famed Ellora caves was carved out of a monolithic rock downwards, i.e. the temple spire first, and then the chambers!

the variables and write the code required for each type of entity. The code dealing with the interaction between two entities must be placed with the entity that is "more involved" in the interaction; but this is of course a subjective judgement. There are various reasons for breaking up the work of programming in this manner. It would seem, intuitively, that if the code organization mirrors the relationships between the entities, then the code might be easier to understand. By breaking up the code into parts, we also make it possible for many programmers to develop the code simultaneously. What we have described above is a very brief and rough overview of the OOP methodology. We will provide more ideas and rationale in the next chapter and chapters 25 and 26.

In this chapter, we will introduce the notion of a *structure*, which is a very basic facility needed for organizing code as mentioned above. Consider the library example. Suppose we wish to write the code associated with books. For this, variables will be needed to store data such as the title, name of the author, price as well as state information such as whether the book has been borrowed, and if so by whom. Presumably, we would like to organize variables related to a single book into a group and give the group a name. By using the group name we would be conveniently able to refer to all the information related to a book, say for the purpose of passing the information to a function. Notice that the variables we wish to group together have different types, so an array will not do. A *structure*, as we will discuss in this chapter, provides us what we want: it allows us to group together data of different kinds into a single collection which we can collectively refer to by a single name. Simply put, for each entity in our program we will have a structure which will hold the data associated with the entity.

We begin by discussing the basic ideas of structures. We will show several examples. We will also revisit the taxi-dispatch problem of Section 14.2.6 and show how its program can be improved using a structure for representing queues. We will also discuss a structure for representing three-dimensional vectors. The notion of a structure to group together variables is inherited into C++ from the C language. However, the notion has evolved substantially in C++. In particular, we can also associate code with a structure, by using so called *member functions*. As you might guess, member functions fit perfectly what we said earlier: they can be used to code the interactions that the entity represented by the structure is involved in. We discuss member functions in Section 17.5.

Member functions are only one new feature introduced into structures in the C++ language. There are several others; these we will discuss in Chapter 18.

17.1 BASICS OF STRUCTURES

As mentioned above, a *structure* is a collection of variables. The variables in the collection are said to be *members* of the structure. You can define different types of structures, as per your need. For example, to store information about books, you might define a structure of type `Book`; you can specify that every structure of type `Book` should contain members to store its name, title, price and so on.

A structure type can be defined using a `struct` statement as follows:

```
struct structure-type {
  member1-type member1-name;
  member2-type member2-name;
  ...
};
```

This statement says that the name `structure-type` will denote a type of collection of variables or *members* whose types and names are as given. The names for structure types or members must be

identifiers (Section 3.1.1), but it is often customary to capitalize the first letters of the names of structure types, which is a convention we will follow.

As an example, here is how we might define a structure type `Book` to store information about a book.

```
struct Book{
  char title[50];
  char author[50];
  double price;
  int accessionNo;
  bool borrowed;
  int borrowerNo;
};
```

Note that a structure type definition does not by itself create variables or reserve space. But we can use it to create variables. Variables thus created are also called *instances*, or *objects*. Here is an example.

```
Book pqr, xyz;
```

This statement is very similar to a statement such as `int m,n;`. The statement `int m,n;` creates variables `m,n` of type `int`. Likewise, the statement `Book pqr, xyz;` also creates variables `pqr,xyz`, of type `Book`. As you might expect, each of these variables is used to hold the associated collection of members. Thus, each variable is allocated as much space as is needed to store the collection. Assuming 4 bytes are used to store an `int` and 8 for a `double`, we will need 16 bytes to store the members `accessionNo`, `borrowerNo`, and `price`, and 50 + 50 bytes to store the members `title` and `author`. A `bool` data type will typically be given 1 byte. So a total of 117 bytes has to be reserved each for `pqr` and `xyz`. The number of bytes that effectively get used might be larger, because there may be restrictions, e.g. on many computers it is necessary that the starting address of a variable must be a multiple of 4.

The word *structure*, or its short form `struct`, is often used to denote (a) a specific variable of a specific structure type, e.g. the variable `xyz` above, or (b) a specific structure type, e.g. `Book` as defined above, or (c) the entire category of variables of any structure type. This may sound confusing, but it is really similar to how we use language in everyday life. For example, in everyday conversation the word *flower* might mean the specific lotus which you have just plucked, or a specific type of flower, as in "a lotus is a flower", or the entire category of flowers, as in "every flower is pretty in its own way". The precise meaning will be clear from the context.

A member of a structure variable can be referred to by joining the variable and the member name with a period (*dot* operator), e.g. `xyz.accessionNo`. Such references behave like variables of the same type as the member. Thus, given the definitions of `xyz` and `pqr` above, we may write:

```
xyz.accessionNo = 1234;
cout << xyz.accessionNo + 10 << endl;
cin.getline(pqr.title,50);
```

The first statement will store the number 1234 in the member `accessionNo` of the variable `xyz`. The second statement calculate the sum of `xyz.accessionNo`, in which we just stored 1234, and the number 10, and print the result. Thus, this statement will cause 1244 to be printed. In the third

statement, the reference `pqr.title` refers to the first of the two `char` arrays in `pqr`. Just as we can read a character string into a directly defined `char` array, so can we into this member of `pqr`.

We can initialize structures in a manner similar to arrays. Assuming `Book` defined as above we might write

```
Book b = {"On Education", "Bertrand Russell", 350.00, 1235, true,
5798};
```

This will copy elements of the initializer list to the corresponding members in the structure. In Section 18.1 you will see how to define other ways of initializing structures.

Here is a structure for representing a point in two dimensions.

```
struct Point{
  double x, y;
};
```

We may create instances of this structure in the manner described before, i.e. by writing something like `Point p1;`. We are allowed to have one structure be contained in another. Here, for example, is a structure for storing information about a circle

```
struct Disk{
  Point center;
  double radius;
};
```

We did not call this `Circle` so as to distinguish it from the `Circle` shape in Simplecpp. Now the following natural program fragment is legal.

```
Disk d1;
d1.center.x = 0.5;
d1.center.y = 0.9;
d1.radius   = 3.2;
```

We could also have accomplished this by initialization:

```
Disk d1 = {{0.5,0.9},3.2};
```

One structure can be copied to another using an assignment. For example, assuming `d1` is as defined above, we can further write

```
Disk d2;
d2 = d1;
```

This causes every member of `d1` to be copied to the corresponding member of `d2`. In a similar manner, we could also write

```
Point p = d1.center;
d2.center = p;
```

The first statement copies every member of `d1.center` to the corresponding members of `p`. The second copies every member of `p` to the corresponding member of `d2.center`.

We finally note that variables can be defined in the same statement as the definition of the structure. For example, we could have written

```
struct Disk{
  Point center;
  double radius;
} d1;
```

which would define the `struct Disk` as well as an instance `d1`.

You can have `const` structures if you wish. For example, you may write

```
const Disk d3 = {{1,2},3};
```

As you might expect, you will not be able to subsequently modify the values of the members of `d3`.

17.1.1 Visibility of Structure Types and Structure Variables

If a structure type is going to be used in more than one function, it must be defined outside both the functions. The definition must textually appear before the functions in the file.

The rules for accessing structure variables are the same as the rules for variables of the fundamental data types (Section 3.6). Thus, in the block in which a variable is defined, it can be used anywhere following its definition. Also, a name `name` defined in block B might shadow names defined in blocks that contain B, or the name `name` might in turn be shadowed by names defined in blocks contained in block B.

17.1.2 Arrays of Structures

Note that we can make arrays of any data type. For example, we could make an array of circles or books if we wished.

```
Disk d[10];
Book library[100];
```

We can refer to members of the elements of the arrays in the natural manner. For example, `d[5].center.x` refers to the x-coordinate of the center of the fifth circle in `d`. Similarly, `library[96].title[0]` would refer to the starting letter in the `title` of the 96th book in `library`.

17.1.3 Pointers to Structures and ->

The "address of" operator `&` defined in Section 9.8.1 and the dereferencing operator `*` defined in Section 9.8.3 work as you might expect with structures. Here is an example. Assuming the definition of `Disk` as above, we might write

```
Disk d1={{1,2},3}, *dptr;
dptr = &d1;
(*dptr).radius = 5;
```

The first statement declares `dptr` to be of type `Disk*`, i.e. pointer to `Disk`. The second statement stores the address of `d1` in `dptr`. This operation is often described as "the second statements sets

dptr to point to d1". In the third statement, dptr is first dereferenced. Thus *dptr evaluates to d1, and then the radius member of this, i.e. of d1 is set to 5.

C++ provides the *arrow* operator -> where x->y means (*x).y. Hence (*dptr).radius could instead be written as dptr->radius, which you will agree is easier to read.

17.1.4 Pointers as Structure Members

It is possible to have pointers as members of a struct. For example, we might have an alternate way to represent disks as follows.

```
struct Disk2{
  double radius;
  Point *cptr;
}
```

where Point is as before. Thus, we could write

```
Point p1 = {10.0,20.0};
Disk2 d4, d5;
d4.radius = 5;
d5.radius = 6;
d4.cptr = &p1;
d5.cptr = &p1;
```

Thus, we have created d4 and d5 to be disks of radii 5, 6 respectively, both centered at the point p1. Say we wanted to get the x coordinate of the center of d4. For this, we would write d4.cptr->x, which would evaluate to 10.0 as you would expect. Note further that if you write

```
d5.cptr->y = 25;
```

it would change the y coordinate of p1, and hence of the center of both the disks.

17.1.5 Linked Structures

Here is a trickier example.

```
struct Student{
  int rollno;
  Student* bestFriend;
};
```

The intention of the definition should be clear; in each student structure, we wish to store a pointer to the best friend of that student. But for this we have had to use the name Student inside its own definition! However, it does not cause a problem. A pointer to a struct needs the same amount of memory no matter what is inside the struct. Thus, using the new definition we can allocate memory for a Student object easily: we just need to allocate whatever is needed for an int and for a pointer.[2]

[2] The following definition is not allowed, of course:
`struct Student{ int rollno; Student friend; };`
This will require a Student object to contain an internal Student object, and the internal Student object to contain a Student object, and so on. In other words we are defining an infinite object! This is not allowed.

We can use this to link students to their best friends as in the program below.

```
int main(){
  Student s1, s2, s3;
  s1.rollno = 1;
  s2.rollno = 2;
  s3.rollno = 3;
  s1.bestFriend = &s2;
  s2.bestFriend = &s3;
  s3.bestFriend = &s2;

  cout << s1.bestFriend->rollno << endl;
  cout << s1.bestFriend->bestFriend->rollno << endl;
}
```

Thus, after creating instances `s1, s2, s3`, we set them respectively to have roll numbers 1, 2, 3. Then we set `s1`'s best friend to be `s2`, `s2`'s best friend to be `s3`, and `s3`'s best friend to also be `s2`. Thus the first print statement will print 2, while the second will print 3.

We will see detailed examples of such linked structures later.

17.2 STRUCTURES AND FUNCTIONS

It is possible to pass structure variables to functions by value (Section 9.1.1) and by reference (Section 9.7). A function may also return a structure.

It should be noted that the name of a structure variable denotes the content of the associated memory, like ordinary numerical variable names, and unlike the name of an array, which denotes the address of the associated memory. Thus, structures behave like ordinary variables when passed by value and when returned from functions.

We first consider an example for our `Point` structure as defined above. The function below returns a `Point` that is the midpoint of the line joining two given points. It is followed by a main program that uses it.

```
// Point as defined earlier
Point midpoint(Point a, Point b){
  Point mp;
  mp.x = (a.x+b.x)/2;
  mp.y = (a.y+b.y)/2;
  return mp;
}

int main(){
  Point p1 = {0.0, 0.0}, p2 = {100.0, 200.0}, p3;
  p3 = midpoint(p1, p2);
  cout << midpoint(midpoint(p1,p2), p2).x << endl;
}
```

The function calls above execute essentially as per the description in Section 9.1. Consider the call `midpoint(p1,p2)`. First, an activation frame is created. Then the values of the arguments `p1,p2` are copied to the corresponding parameters `a,b` in the activation frame. Then the local variable `mp` is created. Then its members `x,y` respectively are set to the averages of the corresponding members of `a,b`. Thus `mp.x, mp.y` will get the values 50, 100. Then, `mp` will be returned. Note that returning a structure means copying its *value* to the main program. To receive this value, in the main program a temporary variable of type `Point` will be created. The main program (or in general the calling program) can do anything with this variable except modifying it. In the second statement of the main program, we are merely copying the value of the temporary variable to the variable `p3`. In the last statement, the call `midpoint(p1,p2)` will return the point (50,100). The temporary variable which holds this is then passed as an argument to another call to `midpoint`. This call will return the midpoint of the points (50,100) and (100,200). Thus, it will return the point (75,150). This point will be stored in a temporary variable, and finally we take its `x` member, which gets printed. Thus, 75 will get printed.[3]

We can also use call by reference (Section 9.7). Suppose we want to shift a given point by some amount `dx,dy` in the x, y directions respectively. Then the following function is natural to write.

```
void move(Point &p, double dx, double dy){
  p.x += dx;
  p.y += dy;
}
```

Notice that we are passing the first argument, the point, by reference. Thus, the point `p` in the body will be deemed to refer to same variable that is passed as the first argument in the calling program. Thus, the `x,y` members of that variable will get modified.

It is often desirable to pass structures to functions by reference even if we don't want them modified. This is because when passed by value, the entire structure must be copied. Copying takes time. This can be a significant overhead for a large structure such as `Book` defined above. In addition, the activation frame of the called structure will also have to have memory allocated to store the structure. This increases the total memory requirement of the program. Thus we may choose to write the `midPoint` function using references.

```
Point midpoint2(Point &a, Point &b){ // also see midPoint3 below
  Point mp;
  mp.x = (a.x+b.x)/2;  mp.y = (a.y+b.y)/2;
  return mp;
}
```

This will indeed cause the points to be passed by reference, thus avoiding the need for copying.

There is, however, a subtle point to be noted about reference parameters. As we noted in Section 9.7.1, A compiler expects that a reference parameter will be modified, so passing a constant argument is an error. So it would indeed be an error if we were to call `midpoint2` with points declared `const`. We noted above that the result of a call such as `midPoint2(p1,p2)` is a temporary variable created by the compiler. This temporary variable is not expected to be modified, i.e. it is considered to

[3] Note that temporary variables are created also with ordinary numerical variables. For example, when you write `a = z + sin(x)*y`, temporary variables will be created to hold the value of `sin(x)`.

have the attribute `const`. So such a temporary variable also cannot be passed to another call in which a reference argument is expected. Thus, a call such as

```
midpoint2(midpoint2(p1,p2),p2)
```

will be flagged by the compiler as an error. This seems inconvenient; we would like to make such calls. To facilitate this, we can declare reference parameters to be `const`, as follows.

17.2.1 `const` Reference Parameters

If we are passing a parameter by reference only to avoid copying it and we do not intend to modify it, we can signal our intention in the code by marking the parameter as `const`. Thus the `midPoint2` function is better written as follows.

```
Point midpoint3(Point const &a, Point const &b){
  Point mp;
  mp.x = (a.x+b.x)/2;  mp.y = (a.y+b.y)/2;
  return mp;
}
```

This has two benefits. First, because the parameters to `midpoint` are marked `const`, any reader will immediately realize, without having to see the entire code, that the arguments to the function will not change. Thus, the readability of our code is improved.

The `const` keyword serves another important purpose too. The corresponding arguments can be `const` structures if we so wish. Thus, the call

```
midpoint3(midpoint3(p1,p2),p2)
```

will now compile fine. The first argument to the outer `midpoint3` call is a temporary constant created by the compiler. But this is fine since the corresponding parameter is marked `const`.

17.2.2 Passing Pointers to Structures

Note finally that you can pass structures to functions using a pointer and then dereference the pointer in the body to access the members of the structure. But it is considered better to use references (`const` if appropriate).

17.3 TAXI DISPATCH REVISITED

Let us revisit the taxi-dispatch problem of Section 14.2.6 from the point of OOP. In that problem, we mimicked a blackboard on which IDs of waiting drivers would be written in real life. The blackboard was not present in the statement of the problem. But it was an important entity in the solution of the problem and hence, it is a good idea to use a `struct` to represent it. We do this next.

The blackboard was really doing the work of a *queue* in which we put in IDs of waiting drivers. The term *queue* has a real life meaning: people wait in it and people leave from it in the order in which they arrive. Likewise, IDs enter our blackboard and then leave in the same order. So we will call our `struct` a `Queue`, which suggests its function, rather than calling it a blackboard. Inside the struct we will have as members all related variables, `driverID, front, nWaiting`. Note, however, that queues can be used to represent other entities besides waiting drivers, so for the array which held the

IDs of the drivers we will use the name `elements` rather than `driverID`. Likewise, we will use the name `QUEUESIZE` to denote the size of the array rather than the name `MAXWAITING`.

```
struct Queue{
  int elements[QUEUESIZE], nWaiting, front;
};
```

`QUEUESIZE` should be defined earlier, say as

```
const int QUEUESIZE = 100;
```

A queue is involved in two kinds of interactions: insertion and removal of elements. So we will have functions which do this.

```
bool insert(Queue &q, int value){
  if(q.nWaiting >= QUEUESIZE) return false;  // queue is full
  q.elements[(q.front + q.nWaiting) % QUEUESIZE] = value;
  q.nWaiting++;
  return true;
}
bool remove(Queue &q, int &item){
  if(q.nWaiting == 0) return false; // queue is empty
  item = q.elements[q.front];
  q.front = (q.front + 1) % QUEUESIZE;
  q.nWaiting--;
  return true;
}
```

Note that we have passed the queue `q` by reference, so that modifications made to it are visible back in the main program. Given these functions, the main program can be written in a nicer manner than in Section 14.2.6.

```
int main(){
  Queue q;
  q.front = 0;
  q.nWaiting = 0;
  while(true){
    char command; cin >> command;
    if(command == 'd'){
      int driver;
      cin >> driver;
      if(!insert(q, driver)) cout << "Queue full.
        Cannot register.\n";
    }
    else if(command == 'c'){
      int driver;
      if (!remove(q, driver)) cout << "No taxi available.\n";
      else cout << "Assigning: " << driver << endl;
    }
```

```
    else if(command == 'x') break;
  }
}
```

This main program is easier to understand as compared to the main program of Section 14.2.6. This is because it does not contain much detail about how exactly the waiting IDs are stored in the queue. That detail is moved to the functions `insert` and `remove`. These functions on the other hand are not concerned with how the queue is being used. The two functions together guarantee that so long as the queue is accessed only using these functions, we will get the expected behaviour: (a) whatever we insert into the queue will be given back to us in a first-in-first-out order, (b) we will not insert something when the queue is already full, (c) our accesses to the array `q.elements` will not be out of range. So although our code has become a bit longer, we can see that each piece is easier to understand than the compact main program of Section 14.2.6.

17.4 REPRESENTING VECTORS FROM PHYSICS

In Chapter 19, we will see a program which deals with motion in 3 dimensional space. This program will deal considerably with 3 dimensional vector quantities such as positions, velocities, and accelerations. So we will design a structure which makes it convenient to represent such quantities.

A vector in three dimensions can be represented in many ways. For example, we could consider it in Cartesian coordinates, or spherical coordinates, or cylindrical coordinates. For simplicity, we consider the first alternative: Cartesian coordinates. Thus, we will have a component for each spatial dimension. Clearly, our structure must hold these 3 coordinates. We will call our structure `V3` and it can be defined as

```
struct V3{
  double x,y,z;
};
```

If our program uses three dimensional vectors, very likely it will need to add such vectors, or multiply such a vector by a number. Here is a function to add two vectors. The resulting vector is returned.

```
V3 sum(V3 const &a, V3 const &b){
  V3 v;
  v.x = a.x + b.x;
  v.y = a.y + b.y;
  v.z = a.z + b.z;
  return v;
}
```

Notice that we have made the parameters be reference parameters to avoid copying, but also made them `const` since the parameters are not altered by the function.

Next we have a function to scale up a vector by a numerical factor.

```
V3 scale(V3 const &a, double factor){
  V3 v;
  v.x = a.x*factor;
```

```
  v.y = a.y*factor;
  v.z = a.z*factor;
  return v;
}
```

It is also useful to have a function that computes the length of a vector.

```
double length(V3 const &a){
  return sqrt(a.x*a.x + a.y*a.y + a.z*a.z);
}
```

We can now use these functions to compute the distance s covered by particle having initial velocity u, moving under constant acceleration a after time t, as per the formula $s = ut + \frac{1}{2}at^2$, where it should be noted that s, u, a are vector quantities.

```
int main(){
  V3 u,a,s;
  double t;
  cin >> u.x >> u.y >> u.z >> a.x >> a.y >> a.z >> t;
  s = sum(scale(u,t), scale(a,t*t/2));
  cout << length(s) << endl;
}
```

This will indeed print the distance covered as desired.

17.5 MEMBER FUNCTIONS

We could think of a structure as merely a mechanism for managing data; we organize data into a collection rather than have lots of variables lying around. However, once you define a structure, it becomes natural to write functions which manipulate the data contained in the structures. You might say that once we defined `V3`, it is almost inevitable that we write functions to perform vector arithmetic and compute the Euclidean length. Once we defined `Queue`, it seemed quite natural to define functions `insert` and `remove` as well. Had we defined a structure to represent some other entity, say a book (in a library), we might have found it useful to write a function that performs the record-keeping needed when a book is borrowed.

Indeed, you might consider such functions to be as important to the structure as are members of the structure. So perhaps, should we make the functions a part of the structure itself?

The definition of structures you have seen so far really comes from the C language. In the more modern definition of structures, as it is in the C++ language, the definition of structures has been extended so that it can also include functions. At a high level, the more general definition of a structure is the same as before.

```
struct structure-type {
  member-description1
  member-description2
  ...
}
```

But, now, a member-description may define a *member-function*, in addition to being able to define a data member as before.

We begin with an example. Here is an alternate way to write our struct V3. Following that is a silly main program given to show how the new definition can be used.

```
struct V3{
  double x,y,z;
  double length(){              // member function length
      return sqrt(x*x + y*y + z*z);
  }
  V3 sum(V3 b){                 // member function sum
    V3 v;
    v.x = x + b.x;  v.y = y + b.y;  v.z = z + b.z;
    return v;
  }
  V3 scale(double t){          // member function scale
    V3 v;
    v.x = x*t;  v.y = y*t;  v.z = z*t;
    return v;
  }
  void joker(double q){        // member function, included for fun.
    x = q;
    cout << length() << endl;
  }
};

int main(){
  V3 p = {1.0, 2.0, 3.0};
  cout << p.length() << endl;
  p.joker(5);
  cout << p.x << endl;
}
```

We explain next how this code works, i.e. how member functions are defined and used. In general, the member-description of a member function has the following form.

```
return-type function-name (parameter1-type parameter1,
   parameter2-type   parameter2, ...) {body}
```

As you can see, the definitions of length, sum, scale and joker all fit in this form.

A member function is expected to be called *on* an object of the given structure type, using the same "." operator used for accessing data members. We will use the term *receiver* to denote the object on which the member function is called. A simple example of a member function call is the expressions p.length() in the main program above, with p the receiver. The general form of the call is

```
receiver.function-name(argument1, argument2, ...)
```

The execution of a call happens as follows.

1. The expressions `receiver, argument1, argument2, ...` are evaluated.
2. An activation frame is created for the execution.
3. The values of the arguments corresponding to the non reference parameters among `argument1,...` are copied over to the corresponding parameters.
4. The body of the function is executed. Inside the body, the names of the data members by themselves are considered to refer to the corresponding members of the `receiver`. Inside the body, we can thus read the values of the members of the receiver, or also modify the values if we wish. Note further that inside the body we may also invoke member functions on the receiver, simply by calling them like ordinary functions.

We next discuss how the main program given above will execute. When the call `p.length()` executes an activation frame is first created. Since there are no arguments, there is nothing to be copied. So the body of the function will start executing. In the body, the names `x,y,z` will refer to the corresponding members of the receiver, `p`. Thus, the statement `return sqrt(x*x + y*y + z*z);` will return `sqrt(1.0*1.0 + 2.0*2.0 + 3.0*3.0)`, i.e. $\sqrt{14}$. This will get printed.

When the call `p.joker(5)` executes, an activation frame will again be created. Then the value of the argument, 5, will be copied to the corresponding parameter, `q` in the body of the member function `joker`. Then the assignment `x=q` will cause the member `x` of the receiver, in this case, `p`, to be set to 5. Then there is a call to `length`. Since the function name appears by itself, i.e. not as `r.length()` for any `r`, it is deemed to refer to the receiver itself. Thus, the length of `p` is calculated. Note that the value of `p.x` has changed to 5, and hence the length calculated and printed will be $\sqrt{5 \times 5 + 2 \times 2 + 3 \times 3} = \sqrt{38}$. After this the execution of `p.joker` will finish.

Finally, in the last statement, we will print the value of `p.x`. Note that 5 will get printed, because this is what we set it to in the call `p.joker(5)`.

We next give the code for computing $s = ut + \frac{1}{2}at^2$ using our new `V3` definition.

```
V3 u, a, s;
double t;
cin >> u.x >> u.y >> u.z >> a.x >> a.y >> a.z >> t;
V3 ut = u.scale(t);
V3 at2by2 = a.scale(t*t/2);
s = ut.sum(at2by2);
cout << s.length() << endl;
```

Note that we do not really need to have the additional vectors `ut, at2by2`. Indeed, we could have written out a single long assignment

```
s = u.scale(t).sum(a.scale(t*t/2));
```

The call `u.scale(t)` returns a vector (temporary `V3` object) on which we perform the `sum` operation using as argument the result of the call `a.scale(t*t/2)`.

17.5.1 Reference Parameters and `const`

It is possible to make the parameters to member functions be reference parameters. As noted earlier, by passing a structure by reference, we avoid the overhead of copying it.

It is also good to add `const` qualifiers wherever possible. First, if any of the arguments is not modified by the function, then the corresponding parameter should also be declared `const`. Second,

if the receiver is not modified by the function, we can indicate as much by adding the keyword `const` after the parameter list but before the body. The function `sum` above modifies neither its argument, nor its receiver. Hence it is better written as

```
V3 sum (V3 const &b) const{   // notice the two const keywords
  V3 v;
  v.x = x + b.x;  v.y = y + b.y;  v.z = z + b.z;
  return v;
}
```

We should similarly modify the member functions `scale` and `length`.

17.5.2 Default Values to Parameters

Default values can be given, as for ordinary functions by specifying them as `= value` after the corresponding parameter.

17.5.3 The `this` Pointer

Inside the definition of any ordinary member function, the keyword `this` denotes a pointer to the receiver. Normally, we do not need to use this pointer, because we can get to the members of the receiver by using the names of the members directly. However, it should be noted that we could use `this` too. Thus, we could have written the `length` member function in `V3` as

```
double length(){
  return sqrt(this->x*this->x + this->y*this->y
                              + this->z*this->z);
}
```

But of course this is not really a good use for `this`!

A more interesting use is given in Section 18.5.

17.5.4 Capturing `this` in Lambda Expressions

Bodies of member functions can contain lambda expressions (Section 12.2). You may want to capture the `this` pointer into a lambda expression. You may do so, and the `this` pointer must be captured by value.

Once you capture `this`, you can refer to members of the object by giving the names directly, without prefixing them with `this->`.

You will see an example of this in Figure 27.4.

17.6 CONCLUDING REMARKS

How to organize variables is an important problem in writing large programs. Structures help in this regard by allowing variables to be grouped together heirarchically.

A structure is a collection of variables (or other structures), called members of the structure. The name of a structure refers to the collection as a whole. We can assign one structure to another, or pass a structure to a function by value, or return a structure from a function. All these operations happen by

copying the entire collection. Pointers and references to structures can be used in the natural manner. These are useful because they enable passing structures to functions without copying.

Member functions can also be included in structure definitions. Member functions are expected to contain code that models the interaction of the entities represented by the structure with other entities. Member functions execute in a manner similar to ordinary functions, except that they execute in the context of a receiver object.

It should be noted that the term *structure* as we have used in this chapter is different from the phrase *data structure* which is often used in the programming/algorithm design literature. A "data structure for storing entity X" could include a `struct`, or an array, or several `struct`s and arrays. Also, it might refer to additional aspects, e.g. an array in which you store data in some specific order is a different data structure from an array in which you store data without sorting, or in which you store data according to some different order. The term structure, on the other hand, refers essentially to just the construct `struct` of C++ and similar constructs in other languages.

EXERCISES

1. Define a `struct` for storing dates. Define a function which checks whether a given date is valid, i.e. the month is in the range 1 to 12, and the day is a valid number depending upon the month and the year. Do the same using a member function.
2. Write a function which returns a disk having two given points as the endpoints of a diameter. Assume the definition of the `Disk` structure given in Section 17.1.
3. What is the output when the following code is executed?

```
struct foo{double x,y;};
double f(foo &f1, foo *f2, foo f3){
  double res = (f1.x - f2->x) * (f3.y - f2->y);
  f1.x = 20;
  f2->x = 30;
  f3.x = 40;
  return res;
}
int main(){
  foo p = {0,0}, q = {3,4}, r = {4,3};
  double s = f(p, &q, r);
  cout << s <<' '<<p.x<<' '<<q.x<<' '<<r.x<<endl;
}
```

4. Define a `struct` for storing complex numbers. Define member functions for arithmetic on complex numbers.
5. Define a structure for representing axis parallel rectangles, i.e. rectangles whose sides are parallel to the axes. An axis parallel rectangle can be represented by the coordinates of the diagonally opposite points. Write a member function that takes a rectangle (axis parallel) as the first argument and a point as the second argument, and determines whether the point lies inside the rectangle.

Write a member function which takes a rectangle and `double` values `dx,dy` and returns a rectangle shifted by `dx,dy` in the x and y directions respectively.

6. Assuming the definition of the structure `Point` as given in Section 17.1, define a function

```
Point sectionPoint(Point p1, Point p1, double ratio);
```

`sectionPoint` should return a `Point` on the line segment joining points `p1` and `p2` and dividing the line segment in the ratio `ratio` to 1.

7. Define a function

```
Point centroid(Point p1, Point p2, Point p3);
```

This is expected to return the centroid of the triangle whose vertices are the points `p1, p2` and `p3`. For this, you should note that the x, y coordinates of the centroid are simply the averages of the x, y coordinates of the vertices. Alternatively, you can also use the `sectionPoint` function defined above: the centroid divides any median in the ratio `2:1`, with the larger distance on the side of the vertex.

8. Write a program to answer queries about ancestry. Your program should read in a file that contains lines giving the name of a person (single word) followed by the name of the father (single word). Assume that there are at most 100 lines, i.e. 200 names. After that, your program should receive a name from the keyboard, and print all ancestors of the person, in the order father, grandfather, great grandfather, and so on as known. Assume for simplicity, that all individuals have unique names.
Adapt the ideas from Section 17.1.5.

CHAPTER 18

Object-oriented Programming: Classes

In the last chapter, we advocated defining a structure type for each kind of entity that our program deals with. We gave examples of structure types, `Book` to hold information about books, `Queue` to hold information about queues, `V3` to hold information about three-dimensional vectors. Each structure was designed with a fairly clear idea about how it will be used. For example, consider the `Queue`. We expect that a `Queue` object will be created, and we will set the data members `nWaiting, front` to 0. Subsequently, the functions `insert` and `remove` will be called to insert and remove elements as needed. We also expect that the data members will not be independently modified, i.e. if `q` is an object of type `Queue`, you will not write something like `q.nWaiting = 50;`. The member `q.nWaiting` will change, but this will happen only as a part of the execution of the functions `insert` or `remove`. As designers of a structure, it is perhaps desirable if we clearly state how we expect the structure to be used, and perhaps also *prohibit* inappropriate uses. This is the next idea in Object oriented programming: not only do we represent entities using structures, but we try, as much as possible to ensure that the structure will be used only in the manner we expect. In this chapter, we will see how C++ supports this goal.

The situation is actually quite similar to how electrical devices are designed. For example, a television comes with a control panel on the front (or a remote control) which helps you to control it. If you wish to change the channel or increase the volume, you press the appropriate buttons provided for that purpose. You do not need to open the backside and manipulate any electrical component directly! In a similar manner, the users of the `Queue` structure should be given an *interface* (like the control panel) which tells them the functions using which they can use the structure. Anything else, they should not be allowed to do. Users of `Queue` should be concerned only with the interface just as the users of television sets need only know how to use the control panel. The users of television sets need not know what is inside the box; similarly, the users of `Queue` need not know precisely how the functions provided do their job, so long as they do what they promise. We discussed similar ideas in the contract model for designing functions (Section 9.3).

C++ indeed allows designers to build structures which users must access only through a carefully chosen set of functions (the interface), and whose internal details such as data members are hidden. In fact, C++ allows structure designers glorious control over the entire life cycle of structure

variables. Designers can precisely control how their structure variables will be (a) created, (b) used in assignments, (c) used with different operators, (d) passed to functions, (e) returned from functions, (f) and finally destroyed when not needed. As we have seen in the previous chapter, some default mechanisms are already provided for all of (a), (b), (d), (e), (f). However, in C++, it is possible to customize, i.e. override the default mechanisms by mechanisms of your choice. We will see all this shortly.

We said earlier that the notion of structures was inherited into C++ from the language C. However this feature: the ability to customize is a new feature provided in C++. In fact, the notion of member functions was also not present in the C language. To emphasize the new features, the term *class* was coined by the C++ designers. However, the terms *structure* and *class* are both valid in C++ and are nearly synonymous, except for a minor but suggestive difference which we will discuss in Section 18.7.

18.1 CONSTRUCTORS

A constructor is a special member function that you can write in order to make it easier to initialize, or even *automate* the initialization of a structure. We will show two examples before discussing how constructors work in general.

```
struct Queue{
  int nWaiting, front, elements[QUEUESIZE];
  Queue(){                                    // constructor begins
    front = nWaiting = 0;
  }                                           // constructor ends
}
```

The above code defines the structure `Queue` and a constructor member function for it. Given this definition, suppose we write

```
Queue q1, q2;
```

in the main program. It turns out that this statement will not only allocate memory for `q1, q2`, but also initialize `q1.front`, `q1.nWaiting`, `q2.front` and `q2.nWaiting` all to 0! As you can see, this is very convenient because we will never use a `Queue` without first setting the members `front` and `nWaiting` to 0. Given the above constructor, we don't have to worry about *forgetting* to initialize the members.

Before we explain how constructors work, another basic motivation behind constructors should be noted: as much as possible, outside of a structure definition, we should access only the member functions, and not refer to the data members directly. This is because functions are defined carefully by the designer having considered the proper ways of manipulating the structure. So it is best to not directly access the data members. If data members are not to be accessed outside the definition, then the only way they can be initialized is using a function. A constructor is meant to be such a function.

In our next example, we show two constructors for our class V3 and their use.

```
struct V3{
  double x,y,z;
  V3(double p, double q, double r){     // constructor 1
    x = p;
    y = q;
    z = r;
```

```
  }
  V3(){                                    // constructor 2
    x = y = z = 0;
  }
  // description of other member functions omitted.
};

int main(){
  V3 vec1(1.0,2.0,3.0);
  V3 vec2;
}
```

The first statement in the main program will create a variable `vec1` of type `V3`, with its `x,y,z` members set to 1.0, 2.0, 3.0 respectively. The second statement will create a variable `vec2` of type `V3` with its members set to 0, 0, 0. As you might guess, the initialization of the two variables has somehow happened using our two constructors respectively. We discuss the precise mechanism of all this next.

In general, a constructor is specified as a member function of the same name as the structure type. Further, there is no return type. Here is the general form.

```
structure-type (parameter1-type parameter1,
                parameter2-type parameter2, ...){ body }
```

You can specify as many constructors as you want, so long as the list of parameter types are different for each constructor.

Whenever a variable of type `structure-type` is defined in the program, memory is allocated for the variable, and then an appropriate constructor gets called on the created variable to initialize it. Which constructor gets called depends upon whether the name of the variable in the definition is followed by a list of arguments. If there is an argument list, then a constructor with a matching list of parameters is selected. Thus, in case of our definition of `vec1` above, there is a list with 3 double arguments. Hence, constructor 1 is selected. If no argument list is given following the variable name, then a constructor call will be made with no arguments, and so a constructor which can be called without arguments is selected. In the definition of `q1,q2` and `vec2` above, there is no argument list, and hence the constructor taking no arguments (constructor 2 in case of `vec2`) is selected for initializing.

Next the selected constructor is called *on* the variable to be initialized, using arguments as appropriate. In other words, the variable to be initialized serves as the receiver for the call. This call executes like any member function call, as described in Section 17.5. Specifically, an activation frame is created and the argument values are copied to the parameters. Then the body of the constructor is executed, with the receiver being the variable to be initialized.

Let us now see what happens for the statement `V3 vec1(1.0,2.0,3.0);` in our program above. As we said, this would cause constructor 1 to be called on `vec1` using the arguments 1.0, 2.0, 3.0. Thus, in the execution an activation frame is created and the argument values, 1.0, 2.0, 3.0 are copied to the corresponding formal parameters `p,q,r`. Then the body of constructor 1 starts execution. The first statement of the body, `x = p;` sets the `x` member of the receiver, `vec1`, to the value of the parameter `p`. Similarly, the members `y` and `z` are set to the values `q` and `r` respectively. After this the constructor call ends. Thus, at the end, `vec1` will have its members `x,y,z` set to 1.0, 2.0, 3.0 respectively. The statement `V3 vec2;` is executed similarly. It causes the second constructor to be

invoked on `vec2`. As you can see, it will set all 3 members to 0. Similarly, for the initialization of `q1,q2` earlier.

Note that if you want the constructor without arguments to be called, you must not supply any list of arguments; it is not correct to supply an empty argument list. This is because `V3 vec2();` is not the same as `V3 vec2;`. The former means something quite different: it declares `vec2` to be a function that takes no arguments and returns a result of type `V3`, as we discussed in Section 11.2.1.

If one structure is nested inside another, then the constructor executes slightly differently. This and other nuances are discussed in Section 18.1.4.

18.1.1 Calling the Constructor Explicitly

It is often useful to create temporary objects of a given class without bothering to give them names. This is done by writing an expression of the following general form:

```
structure-type (argument1, argument2, ...)
```

We will say this is an explicit constructor call. An explicit constructor call looks like a function call, however, a `structure-type` appears instead of a function name. The expression is evaluated as follows. First, a temporary object of type `structure-type` is created. Then a constructor of `structure-type` of selected, whose parameter list matches the `(argument1, argument2, ...)`. Then the constructor is called on the temporary object using the given arguments. The resulting object is the value of the expression.

As an example, if we have the expression `V3(5,6,7)` in our code, it would evaluate to a temporary `V3` object, in which the members have the values 5, 6, 7.

Using an explicit constructor call, we can write the `sum` member function of Section 17.5 more compactly as follows.

```
struct V3{
  ...members and constructors 1 and 2...
  V3 sum (V3 b){
    return V3(x+b.x, y+b.y, z+b.z);  // explicit constructor call
  }
}
```

18.1.2 Default Values to Parameters

Parameters to constructors can also be given default values. For example, we could have bundled our two constructors for `V3` into a single constructor by writing

```
  V3(double p=0, double q=0, double r=0){
    x = p;
    y = q;
    z = r;
  }
```

Now you could call the constructor with either no argument, or up to three arguments—parameters corresponding to arguments that have not been given will get the default values, in this case 0. Note that if you include our new constructor in the definition, you cannot include any of the constructors we gave earlier. Say you specified the new bundled constructor and also constructor 2. Then a call `V3()`

would be ambiguous, it would not be clear whether to execute the body of constructor 2, or the body of the new constructor in which all 3 parameters are initialized to their specified defaults.

18.1.3 "Default" Constructor

We have said that C++ supplies a constructor, if no constructor is given in the definition of a `struct`. This constructor takes no arguments, and its body is empty. Such a constructor is called a *default constructor*. Actually, the term is used more generally: it has come to mean a constructor that can be called with no arguments, even if such a constructor has been explicitly defined by the programmer. Thus, for `V3` our constructor 2 is a default constructor. Likewise, the bundled constructor defined above would also be a default constructor.

A default constructor is needed if you wish to define arrays of a structure, because each element of the array will be constructed only using the default constructor.

Note that C++ does not supply a default constructor if you give any constructor whatsoever. So if you define a non-default constructor (i.e. a constructor which must take at least one argument), then the structure would not have a default constructor. Thus, you would not be able to create arrays of that structure.

The default constructor is useful also when we nest a structure inside another. We discuss this next.

18.1.4 Constructors of Nested Structures

Suppose a structure `X` has other structures `Y, Z, ...` as members. Then during the call to a constructor for `X`, the constructors of `Y, Z, ...` are called before the body of the constructor of `X` is executed. This happens recursively, i.e. if `Y` in turn has members which are structures.

This rule sounds reasonable, but applying it can sometimes be tricky. Consider the `Point` and `Disk` classes as follows.

```
struct Point{
  double x,y;
  Point(double p, double q){x=p; y=q;}
};
struct Disk{
  Point center;
  double radius;
};
```

Consider what happens when we execute

```
Disk c;
```

As discussed above, the default constructor for `Disk` will be called. Since we did not supply a constructor, C++ will create one for us. Note, however, that this constructor must first construct all the members of `Disk`. To accomplish this, the constructor created by C++ will call default constructors of all the members as well. So in our case, the C++ constructed constructor for `Disk` will call the default constructor for `Point`. But the constructor of our class `Point` takes two arguments, and hence is not a default constructor. Further, because a constructor is given for `Point` C++ will not create any constructors for `Point`. Thus, writing `Disk c;` as above would be a compiler error!

This problem can be solved using *initialization lists*.

18.1.5 Initialization Lists

When a `Point` member is created while constructing a `Disk` object, we must somehow indicate that a two argument constructor must be used. We can do this if we write a constructor for `Disk`. Here is one possible way.

```
struct Disk{
  Point center;
  double radius;
  Disk(double x, double y, double r) : center(Point(x,y)), radius(r)
  {
  // empty body
  }
};
```

The text following the `:` to the opening brace of the body in the above code is an *initilization list*. The initialization list of a constructor says how the data members in the receiver should be constructed before the execution of the constructor itself can begin.

Thus, in this case, the code says that `center` should be constructed using the constructor call `Point(x,y)`, where `x,y` are from the parameter list of the `Disk` constructor. Similarly the member `radius` of the `Disk` being constructed is assigned the value `r`. In general, the initialization list consists of comma separated items of the form

```
member-name(initializing-value)
```

This will cause the member `member-name` to be initialized directly using `initializing-value`. The initializing value can be an expression involving the parameters of the constructor. If the initializing value calls a constructor, then instead of writing out the call, just the comma separated arguments could be given. In our `Disk` constructor, the initialization of `center` happens by calling the constructor `Point`. Thus, the initialization list can be shortened as:

```
center(x,y), radius(r)
```

Note that in our example, all the work got done in using the initialization lists, so the body is empty. Note that we could choose to initialize only some of the members using the initialization list and initialize the others in the body, if we wish.[1]

18.1.6 Constant Members

Sometimes we wish to create structures in which the members are set at the time of the initialization, but not changed subsequently. This *write-once* strategy of programming is very comfortable: it is easier to reason about a program if you know that the values once given do not change.

If we want our `Point` structure to have this property, then we would write it as follows.

```
struct Point{
  const double x,y;
  Point(double x1, double y1) : x(x1), y(y1)
```

[1] Whenever possible you should perform initialization through initialization lists, because it is likely faster.

```
  {} // empty body
}
```

Notice that we have given values to members `x,y` using initialization lists. Thus, the members will be assigned values when the structure is created. Later on, the values cannot be changed; indeed, the compiler will flag an error if you write a statement such as `p.x = 5.0;` where `p` is of type `Point`.

18.2 THE COPY CONSTRUCTOR

C++ allows you to specify how structures are copied when passed to functions by value (Section 9.1.1, and how they can be returned from functions. The model for this is as follows. For every structure, C++ defines by default a so-called *copy constructor*. The copy constructor is used for copying the value of a structure that is being passed by value to a function, and also to copy back the value if a function returns a structure. The default copy constructor merely copies each data member of the source structure to the corresponding member of the destination.

As you have probably guessed, you can yourself redefine the copy constructor to do what you wish. A constructor which takes a single parameter of type reference to the structure type, or constant reference to the structure type is considered to be a copy constructor. If you define such a constructor, that will be used for passing arguments by value and returning values, instead of the automatically generated copy constructor.[2]

Below, we show a copy constructor for our `Queue` structure.

```
struct Queue{
  int front, nWaiting, elements[QUEUESIZE];
  Queue(){                                  // ordinary constructor;
    front = nWaiting = 0;
  }
  Queue(const Queue &source):               // copy constructor
    front(source.front), nWaiting(source.nWaiting){
    for(int i=front, j = 0; j<nWaiting; j++){
      elements[i] = source.elements[i];
      i = (i + 1) % QUEUESIZE;
    }
  }
  ... members insert and remove...
};
```

As you can see, the above implementation of the constructor does not copy the entire member `elements`, but only the relevant portion of it. Clearly, this is more efficient than copying the entire structure.

The main use of the copy constructor will arise in connection with dynamic memory allocation. We will see this in Section 21.3.6. Also see Section 18.8.1.

[2] Note that the source structure whose copy is to be made must be passed by reference! If it is passed by value, it would have to be copied, which would require a call to the copy constructor, and so on..

18.3 DESTRUCTORS

We know that a variable is destroyed when control leaves the block in which the variable is defined. By default, destruction of a variable simply means freeing the memory used for the variable. However, we might wish to take other actions besides freeing up the memory. For this we may specify a *destructor* member function. If a variable of type `T` is being destroyed, then the destructor for `T` is called on the variable, and only then the memory of the variable is freed. The destructor for struct `T` is denoted as `~T`, and is a special member function that takes no arguments and has no return type. Here is an example.

```
struct Queue{
  ... other member definitions as before ..
  ~Queue(){
    if(nWaiting > 0)
      cout <<"Warning: Non-empty Queue being destroyed.\n";
   }
};

int main(){
  Queue q;
  {
    Queue q2;
    q2.insert(5);
  }
}
```

Anytime a `Queue` type variable is destroyed, the function `~Queue` will be automatically called. This will print a warning if a queue containing elements is being destroyed—presumably you might expect that a queue should be destroyed only after all elements in it have been processed.

In the above main program, when control exits the inner block, the variable `q2` will be destroyed. This will cause the destructor to be *automatically* called on `q2`. Since `q2` will not be then empty, the message will be printed. Just before the program terminates, the variable `q` will get destroyed. This will also cause the destructor to be called, on `q`. Since `q` will then be empty, this will not cause a message to be printed.

Note that usually, it is an error to call the destructor explicitly. It will be called automatically whenever a variable is to be destroyed. For now we know only one way a variable can be destroyed: when control leaves a block. In Section 21.1, we will see another way in which variables can be destroyed.

In Section 21.3.5, the more common use of destructors is described. Also see Section 18.8.1.

18.4 OVERLOADING OPERATORS

Consider the struct `V3` that we defined in Section 17.4 for representing three-dimensional vectors. In mathematics, it is natural to add two vectors, the result of which is a third vector, whose components are the sums of the respective components of the first two vectors. To get the sum of two vectors, we defined the member function `sum` in Section 17.4. However, it might be more natural to get the sum of vectors by just using the addition operator. In other words, suppose `v, w` are vectors, i.e. variables of type `V3`. Wouldn't it be nice if we could write `v+w` which would have the same effect as `v.sum(w)`?

This is indeed possible. In this section, we see how it can be done. For this we first need to understand how C++ interprets expressions involving operators such as +.

If @ is an infix operator, i.e. an operator that is customarily written between the operands as in

```
x @ y
```

then C++ considers the above expression to be equivalent to

```
x . operator@ ( y )
```

This is merely a call to the member function named operator@, invoked on the object x, with y as the argument. If such a member function is present, then the expression will be accordingly evaluated! Note that operator is a reserved word.

Here is how you could define the operators + and * to work with our struct V3.

```
struct V3{
  // members and constructors as defined earlier
  V3 operator+ (const V3 &b) const{
    return V3(x + b.x, y+b.y, z+b.z);
  }
  V3 operator* (double t) const{
    return V3(x*t, y*t, z*t);
  }
};
```

Because of the first definition, we can add two V3 objects to produce a new V3 object, identical to what our member sum would have produced. The second definition allows us to multiply a V3 object by a double, exactly mimicking the behaviour of the member function scale. Thus, using these definitions, we can write a much nicer looking main program:

```
int main(){
  V3 u,a,s;
  double t;
  cin >> u.x >> u.y >> u.z >> a.x >> a.y >> a.z >> t;
  s = u*t + a*t*t*0.5;
  cout << s.length() << endl;
}
```

In this, we have assumed that the member function length is also defined as in Section 17.5.

We note that this ability to define operator action on structures should be used with care. Because of our familiarity of mathematics, the interpretation of several operators is very firmly fixed in our minds. If we define operators recklessly, inconsistent with our intuition, it is likely to produce code which will be confusing. Indeed it is recommended that arithmetic operators be redefined only for mathematical quantities, where the operators are used in a similar manner in mathematics. Our definition of + and * are consistent with this recommendation in that the notion of adding mathematical vectors and multiplying a mathematical vector by a number are very standard.

When we define an operator action for a structure, we are said to be *overloading* the operator. The following binary operators can be overloaded.

```
+   -   *   /   %   ^   &   |   <   >   == != <= >=  <<   >>   &&   ||
=   +=   -=   *=   /=   %=   ^=   &=   |=   <<=   >>=   []
```

We will see an example for the operator = in Section 18.5, and for the operator `[]` in Section 21.3.7.

In C++, function calls can also be considered operators! Indeed, C++ treats a function call

```
f(a1,a2,...an)
```

as equivalent to

```
f . operator() (a1, a2, ..., an)
```

Thus, if `f` happens to be a `struct`, for which the member function `operator()` is defined, then it will get called! In other words, you treat `struct` instances just like functions and "call" them if you wish. The `struct` instances which can be called are often termed *function objects*. We will see an example of this in Section 22.3.1.

For overloading unary operators, see Appendix C.3.

18.5 OVERLOADING ASSIGNMENT

The assignment operator is already defined for structures: each member of the right-hand-side structure is copied to the corresponding member of the left-hand-side structure. But you can change that if you wish.

Here is how you might override the default definition of = for our structure `Queue`.

```
struct Queue{
  .. other members as before ..
  Queue & operator=(const Queue& rhs){
    front = rhs.front;
    nWaiting = rhs.nWaiting;

    for(int i = front, j=0; j<nWaiting; j++){
      elements[i] = rhs.elements[i];
      i = (i + 1) % QUEUESIZE;
    }
    return *this;
  }
};
```

We do a member-by-member copy, except that we don't copy the entire `elements` array but just that part of it which is in use. Just as we did for the copy constructor of `Queue`. At the end the function returns a reference to the current object on which the assignment is invoked, i.e. the left-hand side of the assignment as the value of the assignment expression (Section 3.2.6). Thus, we can write multiple assignments in the same statement if we wish, i.e. of the form `q1 = q2 = q3;`. Notice by the way that our code works correctly even for self-assignment, i.e. even if someone writes `q = q;`.

Like the copy constructor, the main motivation for overloading assignment will become clear when we consider dynamic memory allocation, in Section 21.3.4.

18.6 ACCESS CONTROL

Finally, we consider the last step in designing a product: packaging it so that only the control panel shows on the outside and the internal circuitry is hidden.

C++ provides a simple way to hide members. The designer of a structure may designate each member of the structure as either `private`, `public`, or `protected`. Private members can be accessed only inside the member functions of the class, and are not accessible outside the class definition (but also see Section 18.11.2). Public members, on the other hand, are accessible from everywhere. In other words, they can be referred to inside the class definition as needed, but also outside of it. We will explain `protected` members later.

To specify access, we divide the members in the class into groups, and before each group place the labels `public:`, `private:` or `protected:` as we want the members in the group to be considered. You may use as many groups as you wish. For example, we may define the structure queue as

```
struct Queue{
private:
  int front, nWaiting, elements[QUEUESIZE];
public:
  Queue(){...}
  bool insert(int value){...}
  bool remove(int &value){...}
};
```

In this, we have made the data members private, and the function members public. Thus, if we wrote `q.nWaiting = 7` outside the definition, say in the main program, the compiler would flag it as an error. Because the constructor and the functions `insert` and `remove` are public, outside the definition of `Queue` we can only use those.

Making the data members private is a very common idea. Typically, a carefully chosen set of function members is made public.

18.6.1 Accessor and Mutator Functions

Sometimes some data members are directly useful outside of an object. In such cases, it is considered appropriate to make them `private`, and allow access to them by defining accessor and mutator functions.

```
struct Point{
private:
  double x, y;
public:
  double getx(){return x;}     // accessor function
  void setx(double v){x = v;} // mutator function
  double gety(){return y;}     // accessor function
  void sety(double v){y = v;} // mutator function
}
```

With this definition, we could access and modify the coordinates of a point, even though the corresponding data members are private.

Note, however, that the above strategy has an advantage as compared to making the members `x,y` public. Suppose that tomorrow we decide to represent a point using its polar coordinates, say using members `r` and `theta`. Then we can still retain the member functions defined above, but only change the bodies appropriately. For example, the function `getx` would now have to return `r*cos(theta)`. We would have to make changes to the `Point` definition, however, we may not need to change the code that uses `Point`, since the user code does not directly access the data members.

18.6.2 Prohibiting Certain Operations

Note that if we define a copy constructor or an assignment operator with either public, private or protected access control, C++ will not generate the default versions for these. If we make any of these operators private, then it will be equivalent to saying that they cannot be used at all outside the structure definition. Thus, if we make the assignment operator private, then effectively we are forbidding assignment for the structure. If we make the copy constructor private, then we are effectively saying that the structure cannot be passed to a function by value, and also cannot be returned from a function.

In the case of the class `Queue`, there might be some reason to forbid the assignment as well as passing by value. This is because intuitively we might think: an element can only be in one queue, if we make a copy we are perhaps inviting errors. Note that even if we make the copy constructor private, the object can still be passed to functions, but only by reference.

18.7 CLASSES

A *structure* as we have defined it, except for a minor difference, is more commonly known in C++ as a *class*.

The small difference between the two is as follows. In a structure, all members are considered public by default, i.e. a member that is in a group that is not preceded by an access specifier is considered public. In a class, all members are considered private by default. Thus, you might write

```
class Queue{
  int front, nWaiting, elements[QUEUESIZE];
public:
  Queue(){...}
  bool insert(int value){...}
  bool remove(int &value){...}
...
};
```

This would make members `front, nWaiting, elements` private even though they are not preceded by a `private:` label.

It is more common to use the term *object* to denote instances of a class.

In addition to the features considered in this chapter, there are a number of other features in classes/structures, the most notable of them being *inheritance*, which we will consider in the following chapters.

18.8 SOME CLASSES YOU HAVE ALREADY USED, ALMOST

We should point out that you have already used classes without knowing it.

18.8.1 Simplecpp Graphics

By now you have probably realized that our graphics commands (Chapter 5 and elsewhere) are built using classes. Indeed, the names `Turtle`, `Rectangle`, `Polygon`, `Line` are all names of classes. The commands to create create corresponding objects on the canvas were merely corresponding constructor calls. The various operations we have described on the graphics objects are calls on member functions.

You can perhaps guess how the ability to write our own constructors etc. helps in developing a graphics library. When we execute a statement such as

```
Turtle t;
```

not only must we create a variable, but we must also draw the turtle on the screen. This drawing operation can be done inside the `Turtle` constructor! Similarly, when a graphics object is destroyed, the screen must be redrawn to remove that object from view. This is done as a part of the destructor! In general, there are a number of book-keeping operations needed when dealing with graphics objects, the code for these can be conveniently placed in the constructors, copy constructors, destructors, and other appropriate member functions.

Simplecpp contains an additional class, `Position`, that can be useful. A position merely contains a pair of numbers, typically coordinates, which can be accessed using the `getX` and `getY` member functions. The member functions `getStart` and `getEnd` on a line object will return its starting point and ending point as `Position` objects. Thus if `L` is a line, then `L.getStart().getX()` will return the x coordinate of the starting position of `L`.

18.8.2 Standard Input and Output

Yes, `cin` and `cout` are objects, respectively of class `istream` and `ostream`. But you can have other objects of these classes too, as we see next.

18.9 CLASSES FOR FILE I/O

Your program can read and write files in a manner very similar to how it uses `cin, cout`. Special classes have been provided for this purpose. To access these classes you need to include the header file `<fstream>`.

To read from a file `input.txt`, you must first create an object of class `ifstream`, by writing something like

```
ifstream myinfile("input.txt");
```

As you can see, this is a constructor call with the argument `"input.txt"`. The argument associates the object `myinfile` with the file `input.txt` which must be present in the current working directory. As you might guess, the constructor argument need not be a constant character string. For example, you could read the name (including the full path if appropriate) into a character array and then give that as the argument. The file name could also be given in a `string` object, which will be discussed in Section 22.1.

The class `ifstream` is a *subclass* of the class `istream`. What this really means will be discussed in Chapter 25. For now, this just means that `ifstream` objects behave like `istream` objects such as `cin`. Thus, after `myinfile` is created, you can write statements such as `myinfile >> n;` which will cause a whitespace delimited value to be read from the file associated with `myinfile` into the variable `n`. Note that just like `cin`, `myinfile` will also become `NULL` if there was an error in reading or if the stream has ended. Thus, you can check for these conditions as in Section 13.6.2.

In a similar manner, you can write

```
ofstream myoutstream("output.txt");
```

which will create an object `myoutstream`, of class `ofstream`. The object will be associated with a file `output.txt` which will get created in the current directory. The class `ofstream` is a subclass of `ostream`, and hence you can treat `myoutstream` just like `cout`. Thus you can write values into the associated file using statements such as `myoutstream << n;`.

Here is a program which takes the first 10 values from a file `squares.txt` which is expected to be present in your current directory, and copies them to a file `squarecopy.txt`, which will get created.

```
#include <simplecpp>
#include <fstream>

int main(){
  ifstream infile("squares.txt");
  ofstream outfile("squarecopy.txt");

  repeat(10){
    int val;
    infile >> val;
    outfile << val << endl;
    cout << val << endl;
  }
}
```

The values are also printed out on `cout` which means they will also appear on the screen (unless you redirect standard out during execution). Notice that we have chosen to enter an end of line after each value, while printing to `outfile` as well as `cout`.

When opening a file for output, you can choose to *append* to an existing file. For this you just need to give an additional argument `ios::app` to the constructor.

18.10 HEADER AND IMPLEMENTATION FILES

Quite often, a class (or struct) will be developed independently of the program that uses it, possibly by a different programmer. Thus, we need a protocol by which the code that defines the class can be accessed by code in other files. Following our discussion of functions, it is customary to organize each class `C` into two files: `C.h` and `C.cpp`.

First, some important terms. It is customary to say that the body of each member-function provides an *implementation* of the member function. In fact, the bodies of all member functions together are

said to constitute an implementation of the class itself. When the implementation is given as a part of the class definition, it is said to be given *in-line*. However, when classes are large and developed independently, it is more customary to put the definition of a class `C` without the implementation, into the file `C.h`, the so called header file. The implementation is put into the file `C.cpp`, using some special syntax. If there are any non-member functions closely related to a class, their declarations can also be put in `C.h`, and definitions in `C.cpp`.

Consider the struct `V3` that we have been discussing all along. We will show example files `V3.h` and `V3.cpp` for it. We will make `V3` be a class, and declare the data members `x,y,z` as private, as is customary. The file `V3.h` could be as follows.

```
class V3{
private:
  double x, y, z;
 public:
  V3(double p=0, double q=0, double r=0);
  V3 operator+(V3 const &w) const;
  V3 operator*(double t) const;
  double length() const;
};
```

Note that the default values of the parameters must be specified in the member function declarations, as shown above.

We next show the implementation file `V3.cpp`, which defines the member functions. A definition of a member function `f` appearing outside the declaration of a class `C` is identical to the definition had it appeared in-line, except that the name of the function is specified as `C::f`. The constructor for class `C` will appear as `C::C`, without a return type.

```
#include <simplecpp>
#include "V3.h"

V3::V3(double p, double q, double r){  // constructor
  x = p;   y = q;   z = r;
}
                                       // member functions
V3 V3::operator+(V3 const &w) const {
      return V3(x+w.x, y+w.y, z+w.z); }

V3 V3::operator*(double t) const { return V3(x*t, y*t, z*t); }

double V3::length() const { return sqrt(x*x+y*y+z*z); }
```

It is acceptable if some of the implementations are placed in line in the header file. Typically, small member functions are left in-line in the header file, while the large member functions are moved to the implementation file.

Header guards (Section 11.2.5) can also be used as needed.

18.10.1 Separate Compilation

We can now separately compile the implementation file, and produce, for the class `V3`, the object module `V3.o`, just as in Section 11.2.3. This module, and the header file, must be given to any programmer who uses the class `V3`. Suppose a program using `V3` is contained in the file `user.cpp`, then it must include the file `V3.h`. The program can now be compiled by specifying

```
s++ user.cpp V3.o
```

Other source/object files needed for the program must also be mentioned on the command line, of course.

18.10.2 Remarks

The general ideas and motivations behind splitting a class into a header file and an implementation file are as for functions. In whichever file the class is used, the header file must be included, because the class must be defined. The implementation file or its object module is needed for generating an executable.

18.11 MISCELLANEOUS FEATURES

18.11.1 Another Overloading Mechanism

We can overload an operator `@` also by defining `operator@` as an ordinary (non member) function on appropriate operand types. This is sometimes useful.

We will give two examples of this. We discussed above how you can define the multiplication between a `V3` object and a `double`. Suppose, for convenience we also wish to allow the `double` to be specified as the left-hand operand, and the `V3` object as the right hand operand. In other words, we would like to be able to write `3*v` as well as `v*3`. This can be done by defining the following (non-member) function.

```
V3 operator* (double factor, const V3 & v){
  return v*factor;
}
```

Here, we are assuming that member function `operator*` is already defined earlier, so `v*factor` will be evaluated as per that.

For another example, suppose next that we wish to be able to print `V3` objects on `cout` or to files using `<<`. For this, we just need to overload the operator `<<` to handle `V3` objects as a right hand side operand.

```
ostream & operator<< (ostream & ost, V3 &v){
  ost << v.x <<' '<< v.y <<' '<< v.z <<' ';
  return ost;
}
```

This will enable you to write expressions such as `ost << v`, where `ost` is of type `ostream`, and `v` of type `V3`. Note that `cout` as well as any output stream you create as in Section 18.9 has type `ostream`. So the above definition will enable you to print on all such streams. You may wonder how a file stream can have type `ofstream` as well as `ostream`, this will get explained in Chapter 25.

On executing `ost << v`, the members `v.x, v.y, v.z` will get printed on the output stream `ost` separated by single space characters. The function returns `ost` so that you can chain output operations, i.e. so that you can write `ost << v1 << v2;` (Section 13.6.3). Note that throughout we are passing `ostream` objects by reference, because `ostream` objects do not allow copying (Section 18.6.2).

In this, we are assuming that the members `x,y,z` are public. Shortly we consider what to do if the members are not public.

Not all operators can be overloaded as ordinary functions. In particular, the assignment operator and various compound assignment operators, the indexing operator `[]`, the function call operator `()` and the arrow operator `->` can only be overloaded as member functions.

18.11.2 Friends

If you make some members of a `struct` (or a class) private, then they can only be accessed inside the struct definition.

Sometimes this is too restrictive. For example, if you make data members private in struct `V3`, then using what you have seen so far, you will not be able to define the `<<` operator as we did in Section 18.11.1. This is because the function `operator <<` in Section 18.11.1 refers to the members `x,y,z` which we made private.

C++ allows you to overcome this difficulty. You go ahead and define the `operator <<` function as you wish, accessing the private members also. To enable the function `operator <<` to access the private members, you put a line declaring the function as a *friend* in the structure definition.

```
struct V3{
   ...
   friend ostream & operator<< (ostream &ost, const V3 &v);
   ...
}
```

This will declare `operator<<` to be a friend, which means that it is allowed to access the private members of `V3`. In general, the line will read

```
friend function-declaration
```

Note that this is not a function declaration; it is merely a declaration of friendship. The function must be declared/defined separately. One possibility is to declare the function in the header file and define it in the implementation file of the class.

The same function can be a friend of several structures, and several functions be a friend of the same structure. In fact, you can have one structure `A` be a friend of another structure `B`. This way, the private members of structure `B` can be accessed inside the definition of structure `A`. To do this you merely insert the line

```
friend A;
```

inside the definition of structure `B`.

18.11.3 Static Data Members

Suppose you wish to keep a count of how many `Point` objects you created in your program. Algorithmically, this is not difficult at all; we merely keep an integer somewhere that is initialized to 0, and then increment it when we create an object. The question is: how should this code be organized.

First, we need to decide where to place the counter. It would seem natural that the counter be somehow associated with the `Point` type. This can be accomplished using *static data members*, as follows.

```
struct Point{
  double x,y;
  static int counter;            // only declares
  Point(){
    counter++;
  }
  Point(double x1, double y1) : x(x1), y(y1){
    counter++;
  }
};
int Point::counter = 0;          // actually defines

int main(){
  Point a,b, c(1,2);
  cout << Point::counter << endl;
}
```

A static data member is a variable associated with a `struct` type. It is declared by prefixing the keyword `static` to the declaration. A static member does not appear in every object created from the structure type, as ordinary non-static members do. Instead, there is a single copy of each static data member, and you may consider these copies to be shared amongst all objects of the structure type. Static members can be made private, public or protected just like non-static members.

Thus, in the example above, each object of type `Point` will have its own `x` and `y` members. However, there will only be one copy of `counter`. Inside the definition of `Point`, the variable `counter` can be referred to by using the name `counter`. If the static variable is public, then outside the definition it can be referred to by prefixing its name by the `struct` name and `::`. So in this example we have used `Point::counter`.

There is a subtlety associated with static data members. The definition of the structure does not actually create the static data members; a `struct` definition is merely expected to create a *type*, without allocating any storage. Hence, we need the statement marked "`actually defines`" in the code above.

If the class declaration is in a header file, then declaration of the static member must be placed in the declaration in the header file. However the definition must go into the implementation file.

18.11.4 Static Member Functions

You can also have static member functions. For example, in the definition of `Point` above, we may add the definition of the static member function `resetCounter`.

```
  static void resetCounter(){ counter = 0; }
      // note keyword ``static''
```

Static member functions can be referred to by their name inside the structure definition, and by prefixing the structure name and `::` outside the definition. Further, static member functions

are not invoked on any instance, but they are invoked by themselves. So we can write `Point::resetCounter()` in the main program if we wish to set `Point::counter` to 0.

Note that in non-static member functions we use the names of the non-static members by themselves to refer to non-static members of the receiver, i.e. the object on which the non-static member function is invoked. However, for a static member function, there is no receiver. Thus it is an error to refer to non-static members by themselves in the body of a static member function.

18.11.5 Template Classes

Like functions, we can templatize classes as well. The process of defining a class template is very similar. Here is a template version of our `Queue` class.

```
template<class T, int QUEUESIZE>
class Queue{
  int front;
  int nWaiting;
  T elements[QUEUESIZE];
public:
  Queue(){...}
  bool insert(T value){...}
  bool remove(T &value){...}
...
};
```

With this, we can create a queue into which we can store elements of any type! For example, if we wish to have a queue `q` of size 100 in which to store objects of class `V3`, we simply define it by writing:

```
Queue<V3, 100> q;
```

This would effectively cause a `Queue` object to be created, with `V3` appearing in its definition wherever the template argument `T` had appeared, and 100 wherever `QUEUESIZE`.

Note that the template for a class must be present in every source file that needs to use it. So it is customary to place a template in an appropriate header file. A class is generated from a template only when the template is invoked as in the line `Queue<V3, 100> q;` above. Thus there is no notion of separately compiling a template.

18.12 CONCLUDING REMARKS: THE PHILOSOPHY OF OOP

This chapter concludes the introduction to Object Oriented Programming. A number of philosophically important ideas as well as language constructs to implement those ideas were introduced. We present a quick summary.

The first idea was that the organization of a program should reflect the relationships between the entities that are being considered in the program. In this regard, it helps to *aggregate* the data related to an entity into a single object. We considered this in Chapter 17.

As we design representations for entities, we realize that there is the possibility of using these representations in an unsafe manner. To avoid this, OOP recommends that the user be allowed to access the representation only through a set of carefully chosen member functions. The user should be prohibited from directly accessing the data members in the representation. This is an extremely

important OOP idea: *encapsulating* the data members so that they cannot be directly accessed. It is implemented by enabling the designer to specify what is private and public in an object. It must be emphasized, of course, that component design, i.e. class design, is an art. The designer must carefully study all aspects of the entity being represented and then decide what users must be allowed to do and what must be prohibited.

Of course, making things private and prohibiting improper use requires programming effort. You may be tempted to avoid this and leave everything public. If your code is meant for your own, one time use, perhaps it suffices to have no packaging: use a `struct` rather than a `class` and keep everything public. However, good programs tend to evolve. If your program works well, you will inevitably want to make it do more things or give it to others. So in general, it is a good idea to package your data structures well from the very beginning. It will save effort in the long run.

Once the principle of encapsulation is accepted, the need for facilities such as constructors, destructors, and operator overloading naturally emerges. As we saw, once we define a class, then we will want to pass objects of the class to functions, or use them in assignments. But the designer will want to specify the exact manner in which such operations must happen – to ensure safety as well as convenience. Thus, C++ provides features such as constructors. The full power of such features will become more apparent in Chapter 21.

What we have been referring to as "well packaged objects" are more formally called *Abstract Data Types*. A `struct` or a `class` is a data type, but you may wonder why we use the adjective *abstract*. This is because the class implementation is private and hidden from the user. The user is not supposed to care how the implementation actually works, and is expected to work with the specification, which is just an abstract promise given by the developer. The phrase *Data Abstraction* is also used to denote this strategy of revealing only the specification and not revealing the private data members.

EXERCISES

1. We can describe a time duration by specifying three numbers: days, hours and minutes. Define a class to represent time in this manner.
   Define a member function `normalize` which modifies a given time duration so that the number of minutes is less than 60, and the number hours less than 24, i.e. a duration specified as 25 hours should be modified to become 1 day, 1 hour and 0 minutes.
   Define a member function which prints out the duration on `cout`. Overload the + operator so that durations can be added. The result should be a normalized duration, i.e. number of hours should be less than 24 and number of minutes less than 60.
2. Define a class for storing polynomials. Assume that all your polynomials will have degree at most 100. Write a member function `value` which takes a polynomial and a real number as arguments and evaluates the polynomial at the given real number. Overload the `+, *, -` operators so that they return the sum, product and difference of polynomials. Also define a member function `read` which reads in a polynomial from the keyboard. It should ask for the degree d of the polynomial, check that $d \leq 100$, and then proceed to read in the first $d + 1$ coefficients from the keyboard. Define a `print` member function which causes the polynomial to be printed. Make sure that you only print $d + 1$ coefficients if the actual degree is d. Carefully decide which members will be private and which will be public. Overload the >>, << operators so that the polynomial can be read or printed using them.

3. Develop a simple library-management program. At the beginning, your program should read information about books in the library from a file into an array of suitably defined `Book` structures, like the one in Section 17.1. After reading information about books, the program should allow users to issue and return books. For this, you should provide suitable commands. When a user issues/returns a book, the `borrowerno` member in `Book` should store the number of the user who has borrowed the book. Define a suitable structure for holding information about users. This information should also be read from a file at the beginning. Write functions and member functions for doing all this as appropriate. The functions should check that the operations are valid, e.g. a book that is already recorded as borrowed is not being borrowed without first being returned. A command should also be provided to shut down the program. When this command is given, the program should write the current information about books and users into respective files. It should be possible to take these files later as inputs to the program for more library transactions.

4. Define a class for storing complex numbers. Provide 0, 1, 2 argument constructors which respectively construct the complex number 0, a complex number with imaginary part 0 and real part as specified by the argument, and a complex number with real and imaginary parts as specified by the arguments. Overload the arithmetic operators to implement complex arithmetic.

5. Sometimes we don't know the exact values of certain quantities, but only know that the value lies in an interval, say between some numbers L and H. In such cases, we might choose to represent the quantity by the pair of numbers L, H. In other words, we are representing each quantity by the *interval* $[L, H]$. If you have two quantities represented by intervals $[L_1, H_1]$ and $[L_2, H_2]$, then clearly their sum must lie in the interval $[L_1 + L_2, H_1 + H_2]$. Thus, the last interval could be considered to be the sum of the first two intervals. Such a representation is quite useful when there is uncertainty in our knowledge of a quantity.
Define a class `Interval` which enables us to represent quantities which we know lie in a certain interval. Overload the arithmetic operators so that you can perform arithmetic on these quantities while keeping track of the uncertainty. Be careful: although, in general, the uncertainty increases when you perform arithmetic, if you subtract a quantity (however uncertain) from itself, you get 0 with certainty. Your implementation should deal with such possibilities properly. For this, you will have to decide whether two references `R1, R2` are in fact identical. You can do this by writing `&R1 == &R2`.

6. Modify the Queue class so that it is not possible to make a copy of a Queue object, or assign to it. Then write a main program that attempts to make a copy or an assignment. Observe that the compiler will tell you that you are trying to perform an operation that is disallowed.

7. Define a `Car` class for showing a car on the screen. A car should have a polygonal body, and two circular wheels. Add spokes to the wheels. It should be possible to construct cars and move them. When a car moves, the wheels should rotate. Add member functions to scale the car as well.

8. Construct a class `Button` which can be used to create an on-screen button, say a rectangle, which can be clicked. Clearly, you should be able to construct buttons at whatever positions on the screen, with whatever text on them. Also, buttons should have a member function `clickedP` which takes an `int` denoting the position of a click, as obtained from `getClick()`, and determines whether the click position is inside the button. What other member functions might be useful for buttons?

CHAPTER 19

A Project: Cosmological Simulation

It could perhaps be said that the ultimate goal of science is to predict the future. Scientists seek to discover scientific laws so that given complete knowledge of the world at this instant, the laws will enable you to say what each object will do in the next instant. And the next instant after that. And so on. Predicting what will happen to the entire world is still very difficult, partly because we do not yet know all laws governing all objects in the world. Even if we knew all the laws, predicting what happens to a large system is difficult because of the enormous number of computations involved. However, for many systems of interest, we can very well predict how they will behave in different circumstances. For example, we understand the physics of collisions and of the materials used in a car well enough to predict how badly a car will be damaged if it collides against a barrier of certain strength at a certain velocity. The term *simulation* is often used to denote this kind of predictive activity. Indeed, many products are built today only after their designs are simulated on a computer to see how they hold up under different conditions.

In this chapter and chapters 27 and 28, we will build a number of simulations. The simulation in this chapter is cosmological. Suppose we know the state of the stars in a galaxy at this instant. Can we say where they will be after a million years? Astronomers routinely do simulations to answer such questions. We will examine one natural idea for doing such simulations, and then examine the flaws in that idea. We will then see an improved idea, which will still be quite naive as compared to the ideas used in professional programs. We will code up this idea. We will use our graphics machinery to show the simulation on the screen.

19.1 MATHEMATICS OF COSMOLOGICAL SIMULATION

In some sense, simulating a galaxy is rather simple. For the most part, heavenly bodies interact with each other using just Newton's laws of motion and gravitation.[1] As you might recall, the law of gravitation states that, two masses m_a, m_b separated by a distance d attract each other with a force

[1] We will stick to the non-relativistic laws for simplicity.

of magnitude

$$\frac{Gm_am_b}{d^2}$$

where G is the gravitational constant. The vector form of this is also important. If r_a, r_b are the vectors denoting the positions of the masses (with respect to some origin), then the distance between the masses is $d = |r_b - r_a|$. The force on mass m_a is in the direction $r_b - r_a$, and hence we may write the force on mass m_a in vector form as

$$\frac{Gm_am_b(r_b - r_a)}{|r_b - r_a|^3} \tag{19.1}$$

If planets collide, then presumably more complex laws have to be brought in, which might have to deal with how their chemical constituents react. But a substantial part of the simulation only concerns how the heavenly bodies move under the effect of the gravitational force. It is worth noting that such simulations have contributed a great deal to our understanding of how the universe might have been created and, in general, about cosmological phenomenon. Also, the ideas used in the simulations are very general, and will apply in simulating other (more earthly!) physical phenomenon involving fluid flow, stresses and strains, circuits and so on.

Our system, then, consists of a set of heavenly bodies, which we will refer to as stars for simplicity. The state of the system will simply consist of the positions and the velocities of the stars. Suppose we know the initial state, i.e. for each star i we know its initial position r_i and velocity v_i (both vectors). Suppose we want to know the values after some time Δ. Letting r_i', v_i' be the values after time Δ, we may write

$$r_i' = r_i + \overline{v_i} \cdot \Delta \tag{19.2}$$

$$v_i' = v_i + \overline{a_i} \cdot \Delta \tag{19.3}$$

where $\overline{v_i}$ is the average velocity (vector) of the ith particle during the interval $[t_0, t_0 + \Delta]$ and $\overline{a_i}$ is the average acceleration (vector) during the interval. We do not know the average velocities and accelerations, and indeed, it is not easy to compute these quantities. However, the key observation, attributed to Euler, is that if the interval size Δ is small, then we may assume with little error that the average velocity remains unchanged during the interval for the purpose of calculating the position at the end of the interval. Euler's observation is similar to the idea we used in Section 8.2 to integrate $f(x) = 1/x$. Assuming that the average velocity is simply the velocity at the beginning we may write

$$r_i' = r_i + v_i\Delta \tag{19.4}$$

Now, we can easily calculate the new position r_i' for each particle, because we know r_i, v_i. Euler's observation also applies to the acceleration: if the interval is small, then the acceleration does not change much during it. Thus, the average acceleration can be assumed to be the acceleration at the beginning, and we may write

$$v_i' = v_i + a_i\Delta \tag{19.5}$$

We are not given a_i explicitly, but we have all the data to calculate it. The acceleration of the i th star is simply the net force on it divided by its mass m_i. The net force is obtained by adding up the gravitational force on star i due to all other stars $j \neq i$. But we know how to calculate the force exerted

1. Read in the state at time 0: For all stars i read in the position, velocity, mass into r_i, v_i, m_i.
2. Read in Δ, T.
3. For step $s = 1$ to T/Δ:
   // r_i, v_i respectively contain the position, velocity of ith star at time $(s-1)\Delta$.
   a. Calculate acceleration at time $(s-1)\Delta$:
      For all stars i: Set a_i using Eq. (19.6).
   b. Calculate position at time $s\Delta$:
      For all stars i: Set r_i' using Eq. (19.4).
   c. Calculate velocity at time $s\Delta$:
      For all stars i: Set v_i' using Eq. (19.5).
   d. Prepare for next step: Set $r_i = r_i'$, $v_i = v_i'$ for all i
   //r_i, v_i now contain position, velocity of ith star at time $s\Delta$.
4. end for
5. For all stars i: Print r_i.

Fig. 19.1 *First-order Euler algorithm*

by one star on another. Thus, we may calculate the acceleration (vector) of the ith star as

$$a_i = \frac{F_i}{m_i} = \sum_{j \neq i} \frac{G m_j (r_j - r_i)}{|r_j - r_i|^3} \tag{19.6}$$

Thus, we have above a procedure by which we can get the state of all stars at time $t + \Delta$ given their state at time t. Our answers are approximate, but the approximation is likely to be good if Δ is small. Picking a good Δ is tricky; we will assume that we are somehow given a value for it. Suppose now that we know the state of our system at time $t = 0$, and we want the state at time $t = T$. To do this, we merely run T/Δ steps of our basic procedure! In particular, we use our basic procedure to calculate the state at time Δ given the state at time 0. Then we use the state computed for time Δ as the input to our basic procedure to get the state for time 2Δ, and so on. This overall idea is shown in detail in Figure 19.1. In this r_i, v_i, r_i' should be thought of as array elements, i.e. r_i is the ith element of an array r.

It turns out that this method can be extremely slow, because the stepsize Δ must be taken very small to ensure that the errors are small. However, there are many variations on the method which have better running time and high accuracy.

19.2 THE LEAPFROG ALGORITHM

The main source of error in the above algorithm is that we used the velocity and the acceleration at the beginning of each interval instead of the average velocity and average acceleration, as was required in equations (19.2) and (19.3). If the intervals are very small, then we can say that the velocity is nearly constant in the interval, and then the velocity at the beginning is a good estimate of the average value. This is the so called first order estimate. However, if we can get better estimates of the average, then we will be able to use larger intervals. So instead of considering the first order estimate, we consider

the so called second-order estimate: we consider the velocity to be varying linearly with time in the interval. For linear variation, the average value of the velocity is precisely the value at the *midpoint* of the interval! So we could use that value in equation (19.2), rather than the value at the beginning of the interval. This argument applies to acceleration too: if the acceleration increases linearly as a function of time during a certain interval, then the average acceleration is the acceleration at the midpoint of the interval.

The algorithm presented in Figure 19.2 uses second-order estimates for velocity and acceleration as described above to get better accuracy. As in the basic algorithm (Figure 19.1), at the beginning of the execution we read in the initial state, and then Δ and T. As in the basic algorithm, we calculate the positions of the stars at times $\Delta, 2\Delta, \ldots, T$. As in the basic algorithm, in the sth iteration of the loop we move the stars from their positions at time $(s-1)\Delta$ to time their positions at time $s\Delta$. To calculate how much a star moves during this interval, we need to know its average velocity during the interval. As discussed above, we approximate this using the velocity at the middle of the interval, i.e. at time $s\Delta - \frac{\Delta}{2}$. Thus in the sth iteration, we must have available the velocities at time $s\Delta - \frac{\Delta}{2}$.

For the very first iteration, i.e. $s = 1$, we thus need the velocity at time $\frac{\Delta}{2}$. For this in step 3, we first calculate a_i the acceleration for star i at time 0. Next, in step 4, we use a first order estimate (just this once!) to calculate the velocity at time $\frac{\Delta}{2}$. Step 4 stores this value, $v_i + a_i\frac{\Delta}{2}$, in variable $\tilde{v}_i$. Thus, at the beginning of the first iteration $s = 1$, we indeed satisfy the invariant: r_i holds the position at time $(s-1)\Delta = 0$ and $\tilde{v}_i$ holds the velocity at time $s\Delta - \frac{\Delta}{2} = \frac{\Delta}{2}$.

Assuming the invariant, in step 5(a) we have all that is needed to calculate the position of the stars at step $s\Delta$. To prepare for the next iteration, we must calculate the velocity at time $s\Delta + \frac{\Delta}{2}$, knowing the

1. Read in the state at time 0:
For all stars i read in the position, velocity and mass into r_i, v_i, m_i.

2. Read in Δ, T.

3. Calculate acceleration at time 0:
For all stars i: set a_i using Eq. (19.6).

4. Calculate the velocity at time $\frac{\Delta}{2}$:
For all stars i: set $\tilde{v}_i = v_i + a_i\frac{\Delta}{2}$

5. For step $s = 1$ to T/Δ:
// $r_i, \tilde{v}_i$ contain the position, velocity of star i at time $(s-1)\Delta$, $s\Delta - \frac{\Delta}{2}$.

a. Calculate position at time $s\Delta$:
For all stars i: set $r_i = r_i + \tilde{v}_i\Delta$ // r_i updated to position at time $s - \Delta$.

b. Calculate acceleration at time $s\Delta$:
For all stars i: set a_i using Eq. (19.6).

c. Calculate velocity at time $s\Delta + \frac{\Delta}{2}$:
For all stars i: set $\tilde{v}_i = \tilde{v}_i + a_i\Delta$ // $\tilde{v}_i$ updated to velocity at time $s\Delta + \frac{\Delta}{2}$.

// $r_i, \tilde{v}_i$ now contains position, velocity of ith star at time $s\Delta$, $s\Delta + \frac{\Delta}{2}$.

6. end for

7. For all stars i: Print r_i.

Fig. 19.2 *The leapfrog algorithm*

velocity at time $s\Delta - \frac{\Delta}{2}$. The interval considered for the velocity calculation thus starts at $s\Delta - \frac{\Delta}{2}$, and ends at $s\Delta + \frac{\Delta}{2}$. Thus, we need the average acceleration during this interval. We approximate this by the acceleration at the midpoint $s\Delta$ of this interval. But note that in step 5(a) we have calculated the positions of the stars at time $s\Delta$. Thus, we can calculate the accelerations at time $s\Delta$ using Eq. (19.6). This is done in step 5(b). Step 5(c) can thus complete the velocity calculation.

This new algorithm uses the second order estimate (value at midpoint) in all calculations execpt in step 3. So overall the accuracy goes up considerably. You can verify this experimentally, as asked in the exercises. The proof of this is, however, outside the scope of the book.

You should now be able to see why this algorithm is called the *leapfrog* algorithm. The calculation of positions and velocities is at interleaved time instants, i.e. position calculations leapfrog over the velocity calculations in time.

19.3 OVERVIEW OF THE PROGRAM

Let us first clearly write down the specifications. Our input will be positions and velocities of a certain set of stars at time 0. We will also be given a number T. Our goal will be to find the positions and velocities of the stars at time T. We must also show the trajectories traced by the stars between time 0 and time T.

The first question in writing the program is of course how to represent the different entities in the program. The main entity in the program is a star, of course. A star has several attributes, its velocity and position, and its mass. The mass is simply a floating point number. However, the velocity and position both have three components, corresponding to each spatial dimension. As suggested in chapters 17 and 18, we will define a class `Star` to represent stars. Further, we can use our class `V3` from Section 18.10 to represent positions, velocities, and accelerations. The trajectory of a star is also to be shown on the screen. So we will include a graphics object, say a `Circle`, in each star object. When we compute the new position of a star, we should move the `Circle` associated with the star. The star class will need a constructor and some member functions to implement the position and velocity updates.

19.3.1 Main Program

The program follows the outline of Figure 19.2. It reads in T, Δ and then star-related data and creates the star objects. Then the force at time 0 is calculated (step 3 of Figure 19.2), using a function `calculate_net_force`. Then the velocity at time $\frac{\Delta}{2}$ is calculated. For this, the `Star` class has a member function `vStep`, as will be seen later. Then the main loop is entered. Step 5(a) of Figure 19.2 is implemented by another call to `calculate_net_force`, step 5(b) by calls to member function `rStep` in `Star`, and step 5(c) by calls to member function `vStep`. The member function `rStep` performs the animation task, i.e. moving the image of the star on the screen.

```
int main(){
  initCanvas("N body problem",800,800);

  double T, delta; cin >> T >> delta;
  int n;           cin >> n;
  Star stars[n];   read_star_data(stars, n);
```

```
  V3   forces[n];
  calculate_net_force(n,stars,forces);
  for(int i=0; i<n; i++) stars[i].vStep(delta/2, forces[i]);

  for(float t=0; t<T; t+=delta){
    for(int i=0; i<n; i++) stars[i].rStep(delta);
    calculate_net_force(n,stars,forces);
    for(int i=0; i<n; i++) stars[i].vStep(delta, forces[i]);
  }
  getClick();
}
```

Reading in data about the stars is fairly straightforward.

```
void read_star_data(Star stars[], int n){
  float mass, vx, vy, vz,  x,y,z;
  for(int i=0; i<n; i++){
    cin >> mass >> x >> y >> z >> vx >> vy >> vz;
    stars[i].init(mass, V3(x,y,z), V3(vx,vy,vz));
  }
}
```

Forces that stars exert on each other are calculated as follows.

```
void calculate_net_force(int n, Star stars[], V3 forces[]){
  for(int i=0; i<n; i++) forces[i]=V3(0,0,0);

  for(int i=0; i<n-1; i++){
    for(int j=i+1; j<n; j++){
      V3 f = stars[i].forceOf(stars[j]);  // force on i due to j
      forces[i] = forces[i] + f;
      forces[j] = forces[j] - f;
    }
  }
}
```

Note that the force due to star i on star j has the same magnitude as the force due to star j on star i, but opposite direction. So we calculate the force just once, and add it to the total force on star i, and subtract it from the total force on star j. Notice how the V3 class makes it easy to write this function.

These functions can be placed in a file, `main.cpp`. This file should also include the header files `V3.h` of the preceding chapter[2] and `star.h` which we will see next.

[2] As used here, the class `V3` of the preceding chapter must be augmented to handle subtraction, i.e. must also have a member function `operator-`. Accessor functions `getx(), gety()` which return the x, y components are also needed. We leave these as easy exercises.

19.4 THE CLASS `Star`

The header file `star.h` is as follows.

```
class Star {
  Circle image;
  double mass;
  V3 r,v;  // position, velocity
public:
  Star(){};
  void init(double m, V3 position, V3 velocity);
  void vStep(double dT, V3 f);
  void rStep(double dT);
  V3 forceOf(Star &s);
};
```

The data member `image`, of class `Circle`, will be used for producing the graphical animation. The `x,y` coordinates of the position (stored in member `r`) will be used as the position of each body on the screen; you may consider that we are viewing the cosmological system in the z-direction, so that only the x, y coordinates are important. The member `image` will be made to put down its pen, so that the orbit will be traced on the screen, as you will see in the member function `init`, in the implementation file `star.cpp` below.

```
void Star::init(double m, V3 r1, V3 v1){
  mass = m;
  r = r1;
  v=v1;
  image.reset(r.getx(),r.gety(),15);
  image.setColor(COLOR("red"));
  image.setFill(true);
  image.penDown();
}

void Star::vStep(double dT, V3 f){
    v = v + f*(dT/mass);
}

void Star::rStep(double dT){
    V3 d = v*dT;
    image.move(d.getx(),d.gety());
    r = r + d;
}

V3 Star::forceOf(Star &s){
  V3 R = s.r - r;
  return R * (mass * s.mass / pow(R.length(),3));
}
```

This should be self-explanatory.

Fig. 19.3 *Three stars in a figure-of-8 orbit*

19.5 COMPILING AND EXECUTION

The files can be compiled by executing

```
s++ main.cpp star.cpp V3.o
```

where we assume that `V3.h` and `V3.o` from Section 18.10, suitably modified, are in the same directory as `main.cpp`, `star.cpp` and `star.h`.

To execute the program, we need a file containing the data for stars. A sample file `3stars.txt` is as follows.

```
3000
10
3
100 497.00436 375.691247 0  0.466203685 0.43236573   0
100 400 400              0 -0.932407370 -0.86473146  0
100 302.99564 424.308753 0 0.466203685 0.43236573    0
```

This is meant to simulate a three star system for 1000 steps, with $\Delta = 10$. The initial positions and velocities of the stars are given as above. Note that they have been carefully calculated. You can simulate this system by typing:

```
./a.out <3stars.txt
```

The stars will trace an interesting figure-of-8 orbit on which they will chase each other. Figure 19.3 gives a snapshot. The stars have their pen down, and hence the orbits traced are also visible.

19.6 CONCLUDING REMARKS

There are a number of noteworthy ideas presented in this chapter.

The general notion of simulating systems of interest is very important. Given the initial state of a system, and the governing laws, we can in principle determine the next states. However, as we saw, the governing laws can be applied in more or less sophisticated ways, leading to more or less error in the result. Texts on numerical analysis will indicate how the error can be estimated, and will also give even more sophisticated ideas than what we presented.

It is worth pointing out an important similarity between cosmological simulation and numerical integration (Section 8.2). The key point is to note that the displacement of any body is really the integral of the velocity of the body! Thus cosmological simulation is really integration, except that the function to be integrated (the velocity at each step) is discovered as you go along. Thus, you may consider the leapfrog algorithm to be analogous to the midpoint rule considered in Section 8.2.

Our program also illustrates two important program design ideas. First is the idea of building classes to represent the entities important in the program. Clearly, the important entities in our program were

the stars: so we built a class to represent them. But as we noted, there were many vector like entities in the problem: so it was useful to build the class `V3` as well. Finally, note that we did not write one long main program: we identified important steps in the main program and used functions to implement those steps. The functions, even if used just once, more clearly indicated the computational structure of our algorithm.

Finally, a small technical point should also be noted. We needed to create an array of `Star` objects. As we indicated in Section 18.1.3, when an array of objects is created, each object can be initialised only using the constructor which takes no arguments. Hence, we had a `Star()` constructor. But this leaves open the question of how to place data in each object. For this, a common idiom is to provide an `init` member function, as we did. We call the `init` function on each object in the array and set its contents. This idiom will come in useful whenever you need arrays of objects in your programs.

EXERCISES

1. Code up the simple algorithm (Figure 19.1) and compare it to the leapfrog algorithm. Use it to simulate a system consisting of a planet orbiting a star. For small enough velocities, the planet will travel around the star for both methods. You will observe, however, that for the simple algorithm, the orbit will keep diverging for any stepsize, which is clearly erroneous. For the same stepsize, you should be able to observe that the leapfrog orbit does not diverge, or diverges much less.
2. Consider an elastic weightless string of length L tied at both ends to fixed pegs. The string has n marks on it which divide it into $n+1$ pieces, each of length $\frac{L}{n+1}$. A mass m is attached at each mark. Each part of the string has Hooke's constant k, i.e. if the part is stretched by distance Δ, a tension $-k\Delta$ is produced. Suppose one of the masses is gradually moved to some position (x, y), and held there. As a result of this motion, other masses also move to new positions. Now the mass which was held fixed is released. Simulate the motion assuming there is no gravity.
3. Consider a sequence of cars travelling down a single lane road. In a simplistic model, suppose that the cars have the same maximum speed V, and acceleration a and deceleration d. Suppose each car attempts to ensure that it can come to a halt even if the car ahead of it were to stop instantaneously (e.g. because of an accident). Further, assume that the driver is aware of this distance, and slows down if the distance ahead reduces, and speeds up if the distance increases, but only till the speed reaches V. Build a simulation of a convoy of cars which travels along the road on which there are signals present. When a signal turns red, the leading car in the convoy brakes so that it comes to a halt at the signal. Of course, the drivers do not react immediately, but have some response time. Note though that usually it is very easy to see if the car ahead is slowing down, because the tail red light comes on. Incorporate such details into your simulation. Show an animation of the simulation using our graphics commands.

CHAPTER 20

Graphics Events and Frames

You already know that the function `getClick` causes the program to wait for the user to click on the graphics canvas, and then returns a representation of the coordinates of the click position. However, it is possible to interact in a richer manner with the graphics canvas. It is possible for your program to wait for the mouse to be dragged, or a key to be pressed, or similar such events. After the event happens, you can decide what action to take. Using the features that we will discuss in this chapter, you should be able to write very interactive and easy to use programs, and even games.

We begin by describing how you can wait for events, and find out exactly what event has happened. We also discuss the notion of *frames*. This will be useful for giving the appearance of simultaneous movement when there are many moving objects on the canvas. After that we will sketch two applications.

20.1 EVENTS

By *event*, we will mean one of the following:

1. A button on the mouse being pressed.
2. A button on the mouse being released.
3. The mouse being dragged. By this is meant the movement of the mouse with some button pressed.
4. A key being pressed on the keyboard.

If the user performs any of these actions with the graphics canvas active (often called "having focus"), then it is considered an event. Note that the graphics canvas becomes active if you move the mouse over it and press any of the mouse keys, and remains active so long as you keep the mouse within the canvas.

20.1.1 Event Objects

Objects of class `XEvent` are used to hold information about events. They are passed by reference to functions that place information in them about events that have happened; and also to functions that extract information placed in them earlier.

We will not describe the class `XEvent` fully, but will discuss only the relevant details below as needed.

20.1.2 Waiting for Events

The function `nextEvent` has the signature

```
void nextEvent(XEvent &event);
```

A `nextEvent` call causes the program to wait for an event to happen. In this, it is like the statement `cin >> ...`, whereupon the program waits for input to be given. When the function returns, the argument `event` will contain information about the event that has taken place. The program can extract this information and accordingly take actions.

20.1.3 Checking for Events

The function `checkEvent` has the signature

```
bool checkEvent(XEvent &event);
```

A call to `checkEvent` returns `true` if an event has happened since the last call to `nextEvent`. The details of that event appear in `event`. If no event has taken place since the last call to `nextEvent`, or if all events that occurred have already been reported through `checkEevent`, then the function just returns `false`.

It is worth emphasizing that the `checkEvent` function does not wait, unlike `nextEvent`.

20.1.4 Mouse Button-press Events

The function `mouseButtonPressEvent` when called on an event `event` returns `true` iff the event is of mouse-button-press type. Once you know that the event is of mouse-button-press type , you can get additional information about it using the members `event.xbutton.button`, which returns an integer denoting which button was pressed, and `event.xbutton.x` and `event.xbutton.y` which give the coordinates of the mouse at the time the button was pressed. Here is an example.

```
XEvent event;
nextEvent(event);
if(mouseButtonPressEvent(event)){
  cout <<"Mouse button "<< event.xbutton.button
       <<" pressed, at position ("<< event.xbutton.x <<
       <<", "<< event.xbutton.y << endl;
}
```

This code will cause the program to wait until some event happens, and then if the event was the pressing of some mouse button, it will print which button was pressed (i.e. 1, 2 or 3 to denote left, middle, right) and at what canvas coordinates.

20.1.5 Mouse Button Release Events

The function `mouseButtonReleaseEvent` when called on an event `event` returns `true` iff the event is of type mouse-button-release. The member `event.xbutton.button` tells which button was released, and `event.xbutton.x` and `event.xbutton.y` give the coordinates of the mouse at the time the button was released.

20.1.6 Mouse-drag Events

The function `mouseDragEvent` when called on an event `event` returns `true` iff the event was a mouse drag, i.e. the user dragged the mouse after pressing a mouse button. The members `event.xmotion.x` and `event.xmotion.y` give the coordinates of the drag position.

20.1.7 Key-press Events

The function `keyPressEvent` when called on an event `event` returns `true` iff the event was the pressing of a key of the keyboard. The function `charFromEvent` applied to the event returns the `char` denoting the key that was pressed. The members `event.xkey.x` and `event.xkey.y` respectively give the coordinates of the position at which the key was pressed.

Key-press events from some keyboards may not be detected if the "caps lock" or "Num lock" modes are on. Be sure to release these modes first.

20.2 FRAMES

As per what we have described so far, when you execute commands to move graphics objects or somehow change their state, the operations actually happen consecutively, in the order you execute them. But you may wish to create the illusion of simultaneous change. For this the notion of *frames* is provided. You can issue the command

```
beginFrame();
```

which will cause all changes to the screen to be temporarily witheld. If you move an object or change its colour and so on, this will not be shown on the screen. However, when you subsequently issue the command

```
endFrame();
```

the current state of all the graphics objects will be shown. Thus, all changes made to the objects between a `beginFrame()` and the following `endFrame()` will appear to happen at once.

If you only have a few objects on screen, then it may not be necessary to use `beginFrame` and `endFrame`; the updates will appear rapidly and may seem simultaneous. However, if many objects on the screen are changing, framing will ensure that the changes appear simultaneously. A good example of the use of frames appears in Section 26.4.

20.3 A DRAWING PROGRAM

Given below is a simple program which enables you to draw on the canvas. A mouse-button-press event signals the beginning of a line being drawn. Subsequent mouse-drag events show to what position the the user intends to draw. So we simply draw a line from the last point where the mouse was pressed or dragged. For this, we have used the `imprintLine` function rather than create a line and calling `imprint` on it. In our experience, the latter is too slow—the line drawing lags behind the cursor movement. If the user presses the escape key at any time, the program ends.

```
int main(){
  initCanvas("Draw using the mouse", 800,500);
  const char escapekey = '\33';
```

```
  XEvent event;
  short lastx=0, lasty=0;
  while(1){
    nextEvent(event);
    if(mouseButtonPressEvent(event)){
      lastx = event.xmotion.x; lasty = event.xmotion.y;
    }
    if(mouseDragEvent(event)){
      imprintLine(lastx, lasty, event.xbutton.x, event.xbutton.y);
      lastx = event.xbutton.x; lasty = event.xbutton.y;
    }
    if(keyPressEvent(event)){
      if(charFromEvent(event) == escapekey) break;
    }
  }
}
```

20.4 A RUDIMENTARY Snake GAME

Perhaps you are familiar with a game called *Snake*, variations of which are available on many computers and even mobile phones.

The essence of the game is to control a snake that keeps on moving on the screen. Typically, the goal is to steer the snake towards food/prizes, and prevent it from hitting obstacles. There may be variations in which the tail of the snake grows as it eats food. Typically, the snake is represented as a sequence of segments (vertebrae!). The head, or segment 0 has a movement direction, North, East, South or West, and it keeps moving one step in that direction per time step. The subsequent segments follow, i.e. segment i moves to the position of segment $i-1$, for $i \geq 1$. The player can change the direction of the head movement to a new direction, say by typing 'n', 'e', 's', or 'w'.

Here we will develop the core logic of the game, i.e. show the snake on the screen and enable the player to change its direction. The addition of prizes, etc., are left for the exercises.

20.4.1 Specification

We have more or less described the specifications above. Perhaps it is important to stress that the snake must keep on moving if the user does nothing, and change direction only when the user types the appropriate keys.

20.4.2 Classes

The snake is obviously an important entity in this game, and so we should represent it using a `Snake` class.

The snake will have a body which consists of several segments. For simplicity, we will have each segment appear as a circle, So it is natural to have a member array named `body` consisting of `Circles`. We will use a constant `length` to denote the number of segments in the body. We also need to remember the current direction of motion; for this, we use data members `dx, dy` which give the current stepsize by which the snake's head moves in x and y directions.

```
#include <simplecpp>
const int gridsep = 20, xinit = 30, yinit = 20, length = 10,
          npts = 40;

struct Snake{
  Circle body[length];
  int headindex;   // which body element is the head of the snake
  int dx,dy;       // current direction of motion.
  Snake(){         // head at (xinit,yinit) in the coarse grid.
    headindex = 0;
    for(int i=0; i<length; i++)
      body[i].reset((xinit+0.5+i)*gridsep,
                    (yinit+0.5)*gridsep, gridsep*0.5);
    dx = -gridsep; dy = 0;
  }
  void move(char command){
                                    // find direction of motion.
    if(command == 'w')      { dx = -gridsep; dy = 0;}
    else if (command == 'n'){ dx = 0; dy = -gridsep;}
    else if (command == 'e'){ dx = gridsep; dy = 0;}
    else if (command == 's'){ dx = 0; dy = gridsep;}
              // else old direction, do not change dx, dy.
    int tailindex = (headindex +length - 1) % length;
                             // current tail
    body[tailindex] = body[headindex];
                             // current tail now on top of head
    body[tailindex].move(dx, dy);
    headindex = tailindex;  // current tail element becomes head
  }
};

int main(){
  initCanvas("Snake", gridsep*npts, gridsep*npts);
  Snake s;
  while(true){
    XEvent event;
    if(checkEvent(event) && keyPressEvent(event)){
        char c = charFromEvent(event);
        s.move(c);
    }
    else s.move('\0');
          // NULL = continue to move in previous direction.
    wait(0.1);
  }
}
```

Fig. 20.1 *The* `Snake` *class and the main program*

We explain the movement of the snake with an example. Suppose the snake body has 10 segments, numbered 0 through 9, head to tail. Suppose further that currently, `body[i]` represents the `i`th segment of the snake's body. In one step, the head of the snake, segment 0, moves by `dx, dy`. The snake segments 1 through 9 must move into positions earlier occupied by segments 0 through 8. Thus, we must move `body[1]` through `body[9]` into positions earlier occupied by `body[0]` through `body[8]`. But since the segments are visually identical, we don't really need actual movement. We can simply stipulate that from now on `body[0]` through `body[8]` will represent snake body segments 1 through 9. The element `body[9]` which used to represent the tail, will start representing the head of the snake, and it will be moved to the position where we were about to move `body[0]`. Thus we just need to maintain an extra member `headindex` which holds the index of the `body` element currently representing the snake head. In particular, `body[(headindex+i) % length]` will currently represent snake body segment `i`.

Given this, implementing the snake movement is straightforward.

1. Determine the values for `dx, dy` as per the direction typed by the user. For example, if the user types 'w', then the snake must move in the negative x-direction alone. Further, movement must happen a distance `gridsep` at a time. Thus, we must set `dx = -gridsep` and `dy = 0`. If the user does not type one of 'n', 'e', 's', 'w', then we should not modify `dx, dy`.
2. The tail segment should move to where the head should have moved. The tail is at index

   ```
   tailindex = (headindex + length - 1) % length
   ```

   of `body`. So we first move `body[tailindex]` to the same position as `body[headindex]`. This is achieved simply by copying `body[tailindex] = body[headindex]`. Then we move `body[tailindex]` by the calculated valuex `dx, dy`.
3. Finally, we set `headindex = tailindex`, because the element which was representing the tail earlier now represents the head.

This is implemented in the `move` member function of the code for the class snake is given in Figure 20.1.

20.4.3 The Main Program

The main program sets up the canvas and the snake. It then goes into an endless loop in which it checks if the user has typed anything in order to change the direction of the snake. This is done using the function `checkEvent`. If the user has indeed typed a key, then its value is extracted using the `charFromEvent` function. This key is then used as an argument to the member function `move` of the snake, so that the movement happens in the required direction. If the user did not type any key, then we call `move` with `'\0'`, i.e. the NULL character. In this case, and also in the case the user typed in an invalid character, the snake merely continues to move in the direction it was moving earlier.

20.5 CONCLUDING REMARKS

We have discussed the simplest model using which events can be handled. There are more elaborate "event-driven" models for handling events which we will not discuss.

But even our simple model can be used to build interesting applications, having features that you might have seen in professional programs and games. Here is an example. In the drawing program, we used "dragging" to draw a line. Another use of "dragging" is to drag objects around on the screen. In

particular, the user can move the cursor to an object, then press a mouse button, drag the mouse, and finally release the button (drop). This should cause the object to get selected and moved and dropped at the new position.

EXERCISES

1. Modify the drawing program discussed in the chapter so that it "beautifies" what the user draws. Specifically, if the user draws something that nearly looks like a straight line, you should draw it as a straight line. Or a circular arc. Try to come up with some protocols so that the user can draw beautiful pictures without too much effort.
2. Modify the drawing program of the previous exercise so that it does not imprint the lines on the canvas, but merely shows them by creating suitable line and circle objects. Keep track of these objects in suitable arrays. Further, implement the convention that dragging after pressing the left mouse button causes drawing. On the other hand, if the mouse is dragged after the middle button is pressed, it should cause the object on which the mouse currently is to be moved. The movement should stop when the dragging stops.
3. Add prizes/food/walls to the snake game. Also make the snake's length increase by one every time it eats food. Also assign a score to the player depending upon how much food/prizes the snake has eaten, and even just how long the snake has stayed around. Display the score suitably.

CHAPTER 21

Representing Variable-length Entities

Suppose we wish to design a class to represent students. Quite likely, we will need to have a member in it to store the name of the student. How large a character array should we allocate for it? Names can be very small, say "Om Puri", or very long, say "Chakravarti Rajgopalachari". As per what you have learned so far, we will have to allocate an array to store the longest possible name. But this means most of the time much of the array will be unused. We may also need to have a member which stores the grades obtained by the student. We will again be forced to allocate an array long enough to store the grades of the maximum possible number of courses any student could take. A similar situation arises in many other applications. For example, suppose we wish to keep track of polygons drawn on the screen. We will need to allocate a two-dimensional array for this. How large should the dimensions be? We will need to make a guess as to how many polygons there might be, and also the number of sides in the largest of them. This will inevitably lead to wastage of memory.

How to avoid such wastage of memory is the topic of this chapter. We cannot directly use structures/classes/arrays the way we have described them so far. This is because the size of a class/structure/array must be fixed once for all, typically without the knowledge of the size of the entities to be stored in it. The most convenient way of representing entities whose size is not known when we write the program is to use the so called *heap* memory allocation. This is also referred to as dynamic memory allocation. Using this heap memory, we will construct classes which can store entities of different, and even variable sizes, without wasting memory. As an example, we will build a `String` class using which you will be able to store and manipulate variable length text strings efficiently and conveniently. We will discuss the implementation in great detail, so that you can build other similar classes if necessary.

The Standard Library of C++ contains several classes which use heap memory. In fact one of the classes in the library is a `string` class, which can be considered to be an advanced version of the `String` class we discuss in this chapter. We will study the Standard Library including the `string` class in Chapter 22. Chapter 22 will not discuss how these classes are implemented; however, the implementation of the `String` class in this chapter will provide some clues.

21.1 THE HEAP MEMORY

So far we have considered two ways of creating variables. Most commonly, variables are created in the activation frame of some function. In addition, we also have global variables (whose use is discouraged), which are created outside of activation frames, before the program even begins execution.

C++ provides a third mechanism for creating variables. A certain region of memory is reserved for this purpose. This region is called the *heap memory*, or just *the heap*. You can request memory from the heap by using the operator `new`. Suppose `T` is a data type such that each variable of type `T` requires s bytes of storage. Then the expression

```
new call-to-a-constructor-for-T
```

causes a variable of type `T`, or in other words s bytes of memory, to be allocated in the heap. The specified constructor is called on the variable to initialize it. Further, the expression itself evaluates to the address of the allocated variable. To use this allocated variable, you must save the address—this you can do typically by storing it in a variable of type pointer to `T`.

Thus, for the `Book` type as defined in Section 17.1, we could write:

```
Book *p;
p = new Book;
```

The first statement declares `p` to be of type pointer to `Book`. The second statement requests allocation of memory from the heap for storing a `Book` variable. Here, the default constructor will be used, since no arguments are specified. The address of the allocated memory is then placed in `p`. We could of course have done this in a single statement if we wish, by writing `Book *p = new Book;`. The memory allocated can be used by dereferencing the pointer `p`, i.e. we may write

```
p->price = 335.00;
p->accessionno = 12345;
```

to set the price and accession number respectively. You can continue to use the allocated variable as long as you wish, using all the usual operations possible on variables of that type. If at some point you decide you no longer need the variable, you can return the memory back to the heap. For this, you use the `delete` operator as follows.

```
delete p;
```

Here, `p` must be a pointer to the variable whose memory you are returning back to the heap. The delete operation causes the destructor (Section 18.3) to be called on the variable pointed to by `p`, and after that the memory of the variable is returned back to the heap. The benefit of returning the memory back to the heap is that it might get reallocated to you on a subsequent request.

A second form of the new operator allows us to allocate an array in the heap. Again, if `T` is a type then we may write

```
T *q = new T[n];
```

which will allocate memory in the heap for an array of `n` elements of type `T`, and the address of the allocated array would be placed in `q`. We can access elements of the array starting at `q` by using the `[]` operator as discussed in Section 14.3.4. Thus, we could write `q[i]` where `i` must be between 0 and `n - 1` (both inclusive). Note that `T` could be a fundamental data type, or a class. If it is a class,

each object `T[i]` would be constructed by calling the default constructor. You must ensure that such a constructor is available.

The `delete[]` operator is used to return the memory of an array allocated earlier on the heap. So for `q` as defined earlier, we may write

```
delete[] q;
```

This will cause the destructor to be called on each element of the array, and then the memory will be returned to the heap.

Note that once we execute `delete` (or `delete[]`) it is incorrect to access the corresponding address; it is almost akin to entering a house we have sold just because we know its address and perhaps have a key to it. It does not belong to us! Someone else might have moved in there, i.e. the allocator might have allocated that memory for another request. Accessing such a pointer is said to cause a *dangling pointer* error.

We used the term *allocator* above. By this, we mean the set of (internal) functions and associated data that C++ maintains to manage heap memory. These are the functions that get called (behind the scenes, so to say) when you ask for memory using the `new` operator and release memory using the `delete` operator.

21.1.1 Accessibility and Lifetime

A variable allocated in the activation frame, as also a global variable, has a name, and we can access the variable using the name. A variable allocated on the heap does not have a name, but is accessible only through its address! So it is vital that we do not lose the address. Thus, we must not overwrite the pointer containing the address of a variable allocated in the heap, unless we stored the address in some other pointer as well. If we do overwrite a pointer containing the address of a heap variable, and there is no other copy, then we can no longer access the memory area which has been given to us. The memory area has now become completely useless. This is technically called a *memory leak*. We must not let memory leak, we must instead return it back to the heap when not needed, using the `delete` operator so that it can be reused!

Heap-allocated variables are also different in the way they get destroyed. Variables allocated in activation frames are destroyed automatically when control exits the block in which they are created. Hence, such variables are often called *automatic variables*. Heap-allocated variables on the other hand, can only be destroyed by executing an explicit `delete` operation. There is no requirement that the `delete` operation be in the same block of code or even the same function body as the `new` operation used for creating the variable in the first place.

21.1.2 A Worked Out Example

We work out in detail the execution of a program that allocates memory from the heap. This program is given in Figure 21.1 (a). We will assume for sake of definiteness that the heap starts at address 24000. When the execution starts, all the memory in the heap is available.

When the first statement, `int* intptr = new int;` is executed, memory to store a single `int` is given from the beginning of the heap. Since an `int` requires 4 bytes, the 4 bytes with address 24000 to 24003 are reserved, and the address of the first of these bytes, 24000, is returned and stored in `intptr`. Next, memory for an array of 3 characters is requested. For this the next 3 bytes are reserved, starting at 24004. Thus, `cptr` gets the value 24004.

```
int main(){
  int* intptr = new int;
  char* cptr  = new char[3];
  *intptr     = 279;
  cptr[0]     = 'a';
  cptr[1]     = 'b';
  cptr[2]     = '\0'; // ***
  delete intptr;
  delete[] cptr;
}
```

(a)

AF of `main()`
`intptr` : 24000
`cptr` : 24004

(b)

Heap memory	
Address	Content
24000	279
24001	
24002	
24003	
24004	'a'
24005	'b'
24006	0
24007	
24008	
24009	
24010	
24011	
24012	...

(c)

Fig. 21.1 *(a) Program (b) Activation Frame after execution of statement* `***` *(c) Heap area after execution of statement* `***`

The following statement `*intptr = 279;` stores the number 279 into the allocated memory pointed to by `intptr`, i.e. at address 24000. The next three statements store the character string constant `"ab"` into the array pointed to by `cptr`. At this stage of the execution, the memory associated with the program is in two parts: the activation frame which contains the variables `intptr` and `cptr`, and the memory which has been allocated in the heap. Figure 21.1(b) shows the activation frame. The heap is shown in Figure 21.1(c).

The last two statements return the allocated variables back to the heap.

21.1.3 Remarks

It should be clear how we could use the heap to solve the problem discussed at the beginning of the chapter: defining a class to represent students. We sketch a preliminary solution here, but also see Section 21.3.10. Our definition will be something like

```
class Student{
  char* name;
  ...
}
```

The class itself will hold only a pointer to the name. When the length of the name to be stored is known, we allocate an array of that size on the heap, and store its address in the member `name`! Thus, we will use memory efficiently.

In general, we can allocate just as much memory we need from the heap, when we need it, and only when we need it. Thus, no memory need be wasted because of overallocation. Further, as soon as we realize that a certain piece of allocated memory is not needed any longer, we can return it to the heap.

In the following chapters, we will see several problems in which heap memory comes in very handy.

21.2 ISSUES IN MANAGING HEAP MEMORY

It turns out that the heap memory is tricky to use. We have already listed the pitfalls: dangling references and memory leaks. Experience shows that dangling references and memory leaks are responsible for many, many programming errors. To prevent these, it seems that we need (a) a sound strategy for managing memory, (b) a way to implement the strategy without too much effort. We discuss these next.

Let us examine in detail how a dangling reference arises. Suppose `p` is a pointer in which at some time t_1 during execution we stored the address `A` of a variable allocated on the heap. At some later time t_2, we dereference `p` and attempt to access the heap-allocated variable. This access would be a dangling reference if between t_1 and t_2, a `delete q` operation happened, where `q` also contained `A`. Furthermore if we executed `delete q` we must have erroneously decided that the variable at address `A` was no longer needed.

Let us also examine how memory leaks can occur. Simply put, suppose at some time t a pointer `p` is the only pointer in our program that holds the address of a heap-allocated variable. We will have a memory leak if at time t we store some other value into `p`. We will also have a memory leak if for some reason `p` gets destroyed. This could happen if control leaves the block in which `p` is defined, for example.

Based on the above observations, two different memory-management strategies suggest themselves.

- **No-sharing Strategy** The key idea in this is to ensure that at each instant during execution all the pointers in our program point to distinct heap variables. Assume that this holds. Let `p` be a pointer to a heap variable at address `A`. Now we know that no other `q` can contain `A`, and hence no operation `delete q` can possibly delete `A`. Thus, `p` cannot be a dangling reference and we can confidently dereference `p` and access address `A`. We can prevent memory leaks by ensuring that we ourselves execute `delete p` just before we store another value into `p`, or before `p` somehow gets destroyed.

- **Reference-counting Strategy** We allow many pointers to point to the same heap variable, but somehow maintain a count of how many pointers point to each heap variable. As soon as the count drops to 0, we execute `delete` on that variable.

It should be clear, at least at a high level, that both the above strategies are sound, i.e. will prevent memory leaks and dangling references.

The next question is how to implement the chosen strategy automatically. The answer is simple: we will design classes whose member functions adhere to the chosen strategy. Thus if we only use such member functions in our program, we will have no leaks nor dangling references.

In the rest of this chapter, we show how the no sharing strategy can be implemented while building a class for representing text. A similar strategy has already been implemented in the standard library classes. This is discussed in Chapter 22. As you will see, you can use the standard library classes without knowing how they avoid dangling pointers and memory leaks. But if you are curious, the rest of this chapter will give you clues about how the standard library classes work. It will also provide you with a model, should you need to design similar classes.

In Appendix G, we show how the reference-counting strategy can be implemented.

21.3 REPRESENTING TEXT

We show how to use heap allocation for representing text strings. The key idea is that the text itself will be stored in an array which we will allocate on the heap. However, we will strive to automate the allocation and freeing of heap memory.

The representation of text strings will be in the form of a class `String`. As recommended in Chapter 18, we will design the class so that users will access `String` objects through a set of public member functions, whereas the data members will be private. Our hope is that the users do not need to even know that heap memory is being used; yet they should be able to work with text strings conveniently and without getting dangling references or memory leaks.

To make the discussion more concrete, suppose we want an implementation using which we can write a main program like the following.

```
int main(){
  String a,b;
  a = "pqr";
  b = a;              // b should also become "pqr"
  String c = a + b;   // should concatenate a, b.
  c.print();          // should print on screen

  String d[2];        // array of 2 strings
  d[0] = "xyz";
  d[1] = d[0] + c;
  d[1].print();
}
```

As you can see, we want users to deal with strings as if they were fundamental data types. Thus we want to be able to copy string literals into a variable, or the contents of one variable into another. We would like to print strings and also use the operator +, perhaps fancifully, to concatenate strings. Clearly, we will need memory to be allocated in many of these operations. But we will hide all this in the implementation which the user, i.e. the programmer who writes the main program, does not need to know.

Other operations might also be desirable, for example, we might want to pass strings to functions, or have them be returned by functions. We will implement these as well.

21.3.1 Basic Principles

In each `String` object we will only have one data member, `ptr`, which points to the position in the heap where the actual string is stored. We will store the string along with a terminating null character i.e. `'\0'`. This way, we will not need to store the length of the allocated region explicitly. Further, if a `String` variable contains the empty string, we will set its member `ptr` to `NULL`.

No-sharing Strategy As discussed earlier, all pointers to heap variables will be made to contain distinct addresses. In principle, if two `String` variables have the same value, i.e. contain the same text, then potentially we can store a single copy of that text in the heap, and have the `ptr` members

of both the variables point to that copy. This will likely save memory, but is not permitted in the No-sharing strategy.[1] Sharing is allowed in the reference counting strategy discussed in Appendix G.

Our class definition is as below. We will add the member functions as we go along. By defining these properly we will try to create a well packaged `String` class which can be used safely by the user without knowing what is inside.

```
class String{
  char* ptr;
   // will point to address in heap where actual text is stored.
public:
  ... member functions to be added later ...
};
```

21.3.2 Constructor

Initially, when we create a string variable, we want it to hold the empty string. Hence, the constructor is as follows.

```
String::String(){
    ptr = NULL;
}
```

We will maintain the invariant that `ptr` is either `NULL` or points to a variable on the heap storing the string value.

21.3.3 The `print` Member Function

We next discuss the member function `print`. This is very simple.

```
void String::print(){
  if(ptr != NULL) cout << ptr << endl;
  else cout << "NULL" << endl;
}
```

Since `ptr` gives the address from where the string is stored, it suffices to write `cout <<ptr <<endl;`. However, if `ptr` is `NULL`, then we cannot print it, instead we explicitly print out "NULL".

21.3.4 Assignments

We wish to allow two kinds of assignments: (a) storing a character string into a `String` variable, as in the statement `a = "pqr";`, and (b) storing one `String` variable into another, as in the statement `b = a;` in the example program given above.

We consider case (a) first. As discussed in Section 18.5, we can define a member function `operator=` to specify how assignment should work. Since the right-hand side of the assignment is to be of type `char*`, this member function must have a `char*` parameter. In the body of the function we describe what we want to happen to execute the assignment.

[1] This could be considered to be wastage of memory, but we tolerate it to simplify the implementation.

```
String& String::operator=(const char *rhs){
  delete [] ptr;
  ptr = new char[length(rhs) + 1];
  scopy(ptr,rhs);
  return *this;
}
```

We give an example to see how this will work. Suppose `z` is of type `String` and say we have a statement

```
z = "mno";
```

This statement will cause the member function `operator=` to execute, with the variable `z` being the receiver, and the parameter `rhs` being the address of the text string `"mno"`.

Clearly, the assignment should cause the text `"mno"` to be copied into a suitable heap address `A`, and `z.ptr` should be set to `A`.

However, in general, some text, say `"pqr"` would already have been stored in `z` when the control arrives at the assignment statement `z = "mno";`. Thus, `z.ptr` will be pointing to a heap address, say `B`, storing `"pqr"`. Because of the no-sharing principle, `z.ptr` can be the only pointer that contains `B`. So if we overwrite `z.ptr`, the memory at `B` will leak away. To prevent this, we must first delete this memory. This is precisely what the first statement of the code above does. After that we request memory from the heap to store the new value, i.e. as many bytes as the number of characters in `rhs` plus an extra byte to store the null character. In this, we have used the `length` function from Section 15.1.4. After that we copy the text pointed to by `rhs` into the new region. For this we have used the function `scopy` from Section 15.1.4. Finally, we return `*this`. This is done so as to allow multiple assignments in the same statement (Section 18.5).

Next we consider assigning one string to another as in the statement `b = a;`. For this as well we need to do something much like the above. One difference is that the right-hand side of the assignment is a `String` rather than a `char*`. The text that is needed to be copied now comes from taking the `ptr` member of the right hand side, rather than taking the right hand side itself directly. Second, we have to be careful about self-assignments, i.e. our code should work even if someone writes `b = b;`. The easiest way to handle this is to do nothing if the left-hand-side operand and the right-hand-side operand are the same.

```
String& String::operator=(const String &rhs){
  if(&rhs == this) return *this; // self-assignment
  delete [] ptr;
  ptr = new char[length(rhs.ptr) + 1];
  scopy(ptr,rhs.ptr);
  return *this;
}
```

Note that we have made a new copy of `rhs.ptr`, rather than just assign `ptr = rhs.ptr`. This is because of our no sharing principle.

21.3.5 Destructor

We said earlier that if a pointer holding an address of a heap allocated variable gets destroyed, we potentially have a memory leak. Here is an example.

```
{
  String s;
  s = "pqr";
}
```

When control enters the block, the variable `s` is created. This means that memory needed for a `String` struct gets allocated in the current activation frame. The `String` struct contains just a single pointer, and hence memory only for a single pointer is allocated. During the execution of `s = "pqr";`, we will allocate memory on the heap and set `s.ptr` to point to it. When control leaves the block, the rule is that every variable created inside the block is destroyed. For variables of type `struct`, destruction merely means freeing up the memory allocated for the variable *in the current activation frame*, unless a destructor has been defined (Section 18.3). For now, we do not have a destructor defined for `String`. Thus, the memory given to object `s` in the current activation frame will be freed up. Nothing will be done regarding the memory that got allocated on the heap, which `s.ptr` points to. Note, however, that after control leaves the block, `s.ptr` will not be defined. Thus, we will no longer be able to use the memory pointed to by `s.ptr`. Thus, there is a leak!

The cleanest way to solve this problem is define a destructor for `String` and put a `delete[] ptr` operation in it. This suffices.

```
String::~String(){
   delete[] ptr;
}
```

Thus, when control leaves the block discussed above, this destructor is *automatically* called on the variable `s`. Inside the call to the destructor, we will execute `delete[]` on `s.ptr`, thereby returning back to the heap the memory pointed to by `s.ptr`. Only after the destructor finishes its execution will the memory of the struct `s` in the current activation frame be released. Thus, there will be no memory leak.

Remember that the destructor call happens implicitly. So you should never explicitly call the destructor because then it will end up being called twice, with `ptr` being deleted twice, which is erroneous.

21.3.6 Copy Constructor

If we pass a `String` object to a function by value, the object will be copied using the copy constructor of the `String` class. Likewise, the copy constuctor will be used also while returning a `String` object from a function. Of course, C++ provides a default copy constructor for all classes, this makes a member by member copy. The default copy constructor will not work properly for us. Consider the trivial code below.

```
void fun(String t){
}

int main(){
  String s;
  s = "abc";
```

```
  fun(s);
  fun(s);
}
```

This code will produce a runtime error! This has to do with how `s` gets passed to `fun`. Since `s` is being passed by value to `fun`, the default copy constructor will be invoked. It will copy `s.ptr` to `t.ptr`. Note that this immediately violates the no-sharing principle: `s.ptr` and `t.ptr` point to the same heap allocated variable. In any case, let us see what happens as the execution continues. Variable `t` will go out of scope when `fun` returns. Thus, the destructor will get called on it just before returning. Notice that our destructor will execute `delete[] t.ptr;`. This will cause the memory used to store `"abc"` to be returned back to the heap! This is clearly unacceptable, because we really expect `fun` to do absolutely nothing. Note that after the first call to `fun`, `s.ptr` becomes a dangling reference because the memory it points to is not allocated to it. Things get worse when you call `fun` again. Again, `s.ptr` is copied to `t.ptr`, and again `t.ptr` is deleted on return. But now `t.ptr` points to memory that was already returned. In other words, the delete operation is returning memory that is not currently allocated. This signals an error and the program will halt.

The fix should be obvious. We must decide what precise behaviour we want when we pass a `String` to a function or return one from a function. We intuitively think of parameter passing or returning from a function as being similar to an assignment. So our copy constructor must mimic the assignment behaviour we had in `operator=`. The code then is

```
String::String(const String &rhs){
  ptr = new char[length(rhs.ptr)+1];
  scopy(ptr,rhs.ptr);
}
```

The copy constructor is essentially like the assignment operator; however since the left hand side is just being constructed, it can be simpler than an assignment operator. When assigning a `String x` to `String y`, i.e. for `y = x`, we need to perform `delete[]` on `y` because what it points to will no longer be needed. However, if `y` is just being constructed, then we know that its `ptr` member does not point to anything yet. So a delete operation is not needed. Hence, the above code does not contain a `delete[] ptr` operation while the assignment operator of Section 21.3.4 does. Also the assignment operator must check for self-assignment, which is not needed here.

With the above copy constructor included, you should be able to see why the two calls to `fun` in the code discussed earlier will not cause problems: at the beginning of each call some memory will be allocated which will then be deleted at the end. Thus, there will never be the question of deleting unallocated memory.

21.3.7 The `[]` Operator

We would also like to access individual characters in a string by specifying the index. You already know from Section 14.3.4 that `[]` is an operator. We just have to overload it for the `String` class!

```
  char& String::operator[](int i){  // returning a reference.
    return ptr[i];
  }
```

Note that we are returning a reference to `ptr[i]`, not the value of `ptr[i]`. Thus, we can use it on the left-hand side of the assignment statement as well.

21.3.8 The Operator +

For strings `a, b`, we would like the result of the operation `a + b` to be the concatenation of `a, b`. Further, the operation should not change `a, b` themselves.

We could write this as an ordinary function `operator+` taking two `String` arguments; or we could write it as a member function to be invoked on the left-hand side `String`, with the right hand side being supplied as an argument. Below we choose the latter option. Clearly, the function must have return type `String`.

```
String String::operator+(const String &rhs){
  String res;
  res.ptr = new char[length(ptr) + length(rhs.ptr) + 1];
  scopy(res.ptr, ptr);
  scopy(res.ptr, rhs.ptr, length(ptr));
  return res;
}
```

First, we create the `String` variable `res` in which we will return the result. Its member `res.ptr` must point to the concatenation which we must construct. The concatenated string will have length equal to the sum of the lengths of the strings pointed to by the receiver and the argument `rhs`. We will also need an additional byte of memory to append the `NULL` character to the concatenation. Thus, the second statement allocates memory of size `length(ptr) + length(rhs.ptr) + 1`. Then the text in the receiver is copied into `res.ptr` using the function `scopy`. The second `scopy` causes the string at `rhs.ptr` to be appended to whatever is present in `res`. This call assumes the following implementation of a `scopy` function taking three arguments.

```
void scopy(char destination[], const char source[], int dstart=0){
  int i;
  for(i=0; source[i] != '\0'; i++)
    destination[dstart+i]=source[i];
  destination[dstart+i]=source[i];    // copy the '\0' itself
}
```

As you can see, this will copy the `source` string to the `destination` starting at index `dstart`.

The last statement returns `res`. To see exactly how this executes, let us consider the statement `c = a + b;` of the main program. The variable `res` in the activation of `operator+` will hold the pointer to the concatenation just before `operator+` returns. During the return, the variable `res` is copied into a temporary variable in the activation frame of the caller using the copy constructor. It is then assigned to `c` using the assignment operator.[2]

[2] You may think there is a lot of unnecessary copying going on. C++ compilers will typically analyze what is going on and avoid much of it.

21.3.9 Use

Figure 21.2 shows the definition of `String` with all the implementations in-line. We have also included member function `size` which gives the number of characters in the string.

```
class String{
  char* ptr; // will point to address in heap where actual
             // text is stored.
public:
  String(){ ptr = NULL; }
  String(const String &rhs){
    ptr = new char[length(rhs.ptr)+1];
    scopy(ptr,rhs.ptr);
  }
  String& operator=(const char* rhs){
    delete [] ptr;
    ptr = new char[length(rhs) + 1];
    scopy(ptr,rhs);
    return *this;
  }
  String& operator=(const String &rhs){
    delete [] ptr;
    ptr = new char[length(rhs.ptr) + 1];
    scopy(ptr,rhs.ptr);
    return *this;
  }
  String operator+(const String &rhs) const {
    String res;
    res.ptr = new char[length(ptr) + length(rhs.ptr) + 1];
    scopy(res.ptr, ptr);
    scopy(res.ptr, rhs.ptr, length(ptr));
    return res;
  }
  void print() const {
    if(ptr != NULL) cout << ptr << endl;
    else cout << "NULL" << endl;
  }
  int size() const {return length(ptr);}
  char& operator[](int i) const {return ptr[i];}
};
```

Fig. 21.2 *The complete* `String` *class*

Using this, the following function and main program calling it can now be written.

```
String lcase(const String &arg){
  String res = arg;
  for(int i=0; i<res.size(); i++)
    if(res[i] >= 'A' && res[i] <= 'Z') res[i] += 'a' - 'A';
  return res;
}

int main(){
  String a,b;
  a = "PQR";
  b = a;
  String c = a + b;  // should concatenate a, b.
  c.print();         // should print on screen

  String d[2];       // array of 2 strings
  d[0] = "Xyz";
  d[1] = lcase(d[0] + c);
  d[1].print();
  d[1][2] = d[0][1];
  d[1].print();
}
```

This will first print `c`, which will have the value `"PQRPQR"`. Before the second print statement, the program concatenates `"Xyz"` and `"PQRPQR"` and then converts it all to lower case. Thus, `"xyzpqrpqr"` will get printed. After that the program sets the character at index 2 of d[1] to the character at index 1 of d[0]. Thus, the last print statement will print `"xyypqrpqr"`.

21.3.10 A Class to Represent Students

We return to the problem with which we began this chapter: building a class to represent students. Here is how it can be defined.

```
class Student{
  String name;
  Student(const char* n){ name = n;}
  ...
}
```

As you can see, we just define `name` to be of type `String`. We then freely use member functions of `String` inside the definition of `Student`. The key point is that we never need to worry about allocating/deleting memory—all that is handled behind the scenes by the member functions of `String`.

21.4 CONCLUDING REMARKS

The heap memory is very useful in representing variable-length entities. We will see the heap and the operators `new` and `delete` used substantially in the following chapters.

Using `new` and `delete` is tricky. So wherever possible, it is good to encapsulate the operators inside member functions, so as to ensure that their use is *automatically* correct. We showed how this can be done in a `String` class built to store character strings. We showed that the definition was good enough to allow string creation, indexing into strings, concatenating strings, assigning to strings, passing strings to functions, and returning them from functions. Effectively, using our definition, we have an illusion that `String` is a fundamental data type. If you stick to the operations mentioned above, our implementation guarantees that the objects we create will use memory efficiently, without creating dangling pointers or memory leaks. Note that the above operations do not include creating pointers to `String` objects. You could potentially get into trouble if you create `String` pointers. For example, suppose you write the following.

```
String *ptr = new String;                    // Avoid this
```

If this is inside a block, then on exit from the block, the variable `ptr` will get deallocated. As a result, the memory area it points to will leak away. So if you allocate explicitly, it is your responsibility to delete explicitly too. Best to do neither, if possible. Indeed, in many applications, you will be able to use our `String` class without even knowing that there exist heaps or pointers. Heap management will happen behind the scenes. You are expected to sit back and enjoy the convenience, without interfering in the memory management.

Our implementation has a shortcoming though: if two variables of type `String` have the same value, we will keep two copies of the value. This can be improved upon, as discussed in Appendix G.

Many of the ideas we discussed in the implementation of the `String` class are applicable in general, for other classes you may want to write. Many ideas discussed work together. For example, experience shows that when designing classes, constructors, destructors, and copy constructors go together: if you implement one, you very likely have to implement all three! This is often referred to as the "rule of three" in the OOP literature.

21.4.1 Class Invariants

While designing `String`, we made some important decisions early on. In particular, we said that there will be a separate copy in the heap memory of the value stored in each `String` object (No sharing strategy). Such a property that the members of a class possess throughout their lifetime, is sometimes called a class invariant. It is useful to clearly write down such invariants, as you have seen, they guide the implementation of the class.

EXERCISES

1. Consider the following code. Identify all errors in it.

```
int *ptr1, *ptr2, *ptr3, *ptr4;
ptr1 = new int;
ptr3 = new int;
ptr4 = new int;
ptr2 = ptr1;
```

```
ptr3 = ptr1;
*ptr2 = 5;
cout << *ptr2 << *ptr1 << endl;
delete ptr1;
cout << *ptr3 << *ptr4 << endl;
```

The possible errors are: memory leaks, dangling pointers, and referring to uninitialized variables.

2. Overload the << and >> operators so that `String` objects can be printed and read from streams.
3. Suppose you forgot to check for self-assignment in the definition of the assignment operator in Section 21.3.4. What would happen if you wrote `b = b;` where `b` is of type `String`?
4. Suppose you have a file that contains some unknown number of numbers. You want to read it into memory and print it out in the sorted order. Develop an extensible array data type into which you can read in values. Basically, the real array should be on the heap, pointed to by a member of your structure. If the array becomes full, you should allocate a bigger array. Be sure to return the old unused portion back to the heap. Write copy constructors etc. so that the array will not have leaks, etc. Use a *doubling* strategy, i.e. if the currently allocated array is full and you want to extend it, allocate a new array of twice the size. This will ensure that you do not make too many allocation requests.
5. In this assignment, you are to write a class using which you can represent and manipulate sets of non-negative integers. Specifically, you should have member functions which will (a) enable a set to be read from the keyboard, (b) construct the union of of two sets, (c) construct the intersection of two sets, (d) determine if a given integer belongs to a given set, (e) print a given set. Use an array to store the elements in the set. With your functions it should be possible to run the following main program.

```
int main(){
  Set a,b;
  a.read();
  b.read();

  set c = union(a,b);
  set d = intersection(a,b);

  int x;
  cin >> x;

  bool both = belongs(x,d);
  bool none = !belongs(x,c);

  if( both ) {
      cout << x << " is in the intersection ";
      c.print();
  }
  else if (none) cout << x << " is in  neither set." << endl;
  else cout << x << " is in one of the sets." << endl;
}
```

Ensure that you allocate arrays of just the right size for the union/intersection by first determining the size of the union/intersection.

6. Define a class `Poly` for representing polynomials. Include member functions for addition, subtraction, and multiplication of polynomials. Use a no-sharing strategy.

7. Define the modulo operator `%` for `Poly`. Suppose $S(x), T(x)$ are polynomials, then in the simplest definition, the remainder $S(x) \bmod T(x)$ is that polynomial $R(x)$ of degree smaller than $T(x)$ such that $S(x) = T(x)Q(x) + R(x)$ where $Q(x)$ is some polynomial.
The main motivation for writing the modulo operator is to use it for GCD computation later. So it is important to make sure that there are no round-off errors as would happen if you divide. One way around this is to define the remainder $S(x) \bmod T(x)$ to be any $kR(x)$ where k is any number, where $R(x)$ is as defined above. Assuming that the coefficients of the polynomials are integers to begin with, you should now be able to compute a remainder polynomial without fractional coefficients. Hence, there will be no round off either. Of course this has the drawback that the coefficients will keep getting larger. For simplicity ignore this drawback.
Write Euclid's algorithm for a polynomial GCD.

8. Templetize the `gcd` function so that it can work with ordinary numbers as well as polynomials. You will have to define a few more member functions as well as a constructor. Note that `int` is a constructor for the `int` type, i.e. `int(1234)` returns the integer 1234.

9. Consider the following new member function for the class `Poly`:

```
void move(Poly &dest);
```

When invoked as `source.move(dest)`, it should move the polynomial contained in `source` to `dest`, and also set `source` to be undefined. Effectively, this is meant to be an assignment in which the value is not copied but it *moves*. Is it necessary to allocate new memory while implementing `move` in order to adhere to the no sharing strategy?
See if the `Poly` class with the new `move` function will make the GCD programs considered earlier more efficient.

CHAPTER 22

The Standard Library

An important principle in programming is to not repeat code: if a single idea is used repeatedly, write it as a function or a class, and invoke the function or instantiate a class object instead of repeating the code. But we can do even better: if some ideas are used outstandingly often, perhaps the language should *give* us the functions/classes already! This is the motivation behind the development of the Standard Library, which you get as a part of any C++ distribution. It is worth understanding the library, because familiarity with it will obviate the need for a lot of code which you might otherwise have written. Also, the functions and classes in the library use the best algorithms, do good memory management wherever needed, and have been extensively tested. Thus, it is strongly recommended that you use the library whenever possible instead of developing the code yourself.

The library is fairly large, and so we will only take a small peek into it to get the flavour. We will begin with the `string` class, which is very convenient for storing text data. This class is an advanced version of the `String` class of Chapter 21. It is extremely convenient, and you should use it by default instead of using character arrays.

Next we will study the template classes `vector` and `map` which are among the so-called *container* classes supported in the library. They can be used to hold collections of objects, just as arrays can be. Indeed, you may think of these classes as more flexible and more powerful extensions of arrays. We will not discuss how any of these classes are implemented, although you can get some clues from the discussions in Chapter 21 and Section 24.1. But of course, as users you only need to know the specification of the classes, and need not worry about how they are implemented.

As examples of the use of the Standard Library, we program variations on the marks display program of Section 14.2.2. You know enough C++ to solve all these variations, and you have already solved some of them. However, you will see that using the Standard Library, you will be able to solve them with much less programming effort.

At the end of the chapter, we will give a quick overview of the other classes in the standard library. Of these, we will use the `priority_queue` class in Chapter 27.

22.1 THE `string` CLASS

The `string` class is a very convenient class for dealing with `char` data. It is so convenient, that you are encouraged to use the `string` class wherever possible, instead of `char` arrays. To use the string

class you need to include the header file <string>, but note that it will be included automatically as a part of <simplecpp>.

We can create string objects p, q, r very simply.

```
#include <string>   // not necessary if simplecpp is included.

string p = "abc", q ="defg", r;
r = p;
```

The first statement will define variables p, q, r and initialize them respectively to "abc", "defg" and the empty string respectively. The second statement copies string p to string r. When you make an assignment, the old value is overwritten. Notice that you do not have to worry about the length of strings, or allocate memory explicitly.

You can print strings as you might expect.

```
cout << p << "," << q << "," << r <<endl;  //prints ``abc,defg,abc''
```

This will print out the strings separated by commas. Reading in is also simple, cin » p; will cause a whitespace terminated sequence of the typed characters to be read into p. To read in a line into a string variable p, you can use

```
getline(cin, p);
```

Note that you cannot write cin.getline(p) as you might expect from your experience with char* variables. Also see the variation described and used in Section 22.4.1.

The addition operator is defined to mean concatenation for strings. Thus, given the previous definitions of p, q, r, you may write

```
r = p + q;
string s = p + "123";
```

The first statement will set r to "abcdefg". The second will create s and set it to "abc123". The operator += can be used to append.

You can write s[i] to refer to the ith character of string s. Member functions size and length both return the number of characters in the string. There is no notion of a string object being terminated by a null character, unlike the class String we developed in Chapter 21.

Many other useful member functions are also defined. Here are some examples.

```
string v="abcabcd";
v[2] = v[3];           // indexing allowed. v will become "abaabcd".
cout << v.substr(2)    // substring starting at 2 going to end
     << v.substr(1,3) // starting at 1 and of length 3
     << endl;          // will print out "aabcdbaa".

int i = v.find("ab");         // find from the beginning
int j = v.find("ab",1);       // find from position 1.
cout << i << ", " << j << endl; // will print out 0, 3
```

Note that if the given string is not found, then the `find` operation returns the constant `string::npos`, which is a value that can never be a valid index, e.g. some negative value. We can use this as follows:

```
string t;  getline(cin, t);
int i = p.find(t);   // assume p is as defined above.
if(i == string::npos)
  cout << "String: "<< p << " does not contain "<< t << endl;
else
  cout << "String: "<< p << " contains "<< t <<
          " from position "<< i << endl;
```

Finally, we should note that strings have an order defined on them: the lexicographic order, i.e. the order in which the strings would appear in a dictionary. One string is considered < than another if it appears earlier in the lexicographical order. Thus we may write the comparison expressions `p == q` or `p < q` or `p >= q` and so on for strings `p, q` with the natural interpretation.

22.1.1 Passing `string`s to Functions

Since `string` is a class, we can pass it to functions using value, in which case a new copy is passed, or by reference, in which case the called function operates on the argument itself.

22.2 THE TEMPLATE CLASS `vector`

The template class `vector` is meant to be a friendlier, more general variation of one dimensional arrays. To use the template class `vector`, you need to include the header file `<vector>`.

A `vector` can be created by supplying a single template argument, the type of the elements. For example, we may create a vector of `int` and a vector of `float` by writing the following.

```
vector<int> v1;
vector<float> v2;
```

These vectors are empty as created, i.e. they contain no elements. But other constructors are available for creating vectors with a given length, and in addition, a given value. For example, you might write

```
vector<short> v3(10); // vector of 10 elements, each of type short.
vector<char>  v4(5,'a'); // vector of 5 chars, each set to 'a'.
vector<short> v5(v3);    // copy of v3.
```

A vector keeps track of its own size, to know the size you simply use the member function `size`. Thus,

```
v3.size()
```

would evaluate to 10, assuming the definition earlier. You can access the `i`th element of a vector using the subscript notation as for arrays. For example, you could write

```
v3[6] = 34;
v4[0] = v4[0] + 1;
```

The usual rules apply, the index must be between 0 (inclusive) and the vector size (exclusive). You can append additional elements to a vector using the member function `push_back`.

```
v3.push_back(22);
v1.push_back(37);
```

The argument to the function is appended to the receiver. Thus, the first line above would append the number 22 to `v3`, causing its length to increase to 11 from the earlier 10 as defined above. The second statement would append 37 to `v1`. As defined above, `v1` was empty; thus at the end of this statement it would have length 1.

A whole bunch of operations can be performed on vectors. For example, unlike arrays, you can assign one vector to another. So if `v,w` are vectors of the same type, then we may write

```
v = w;
```

which would make `v` be a copy of the vector `w`. The old values that were contained in `v` are forgotten. This happens even if `v,w` had different lengths originally. You should realize that although the statement looks very small and simple, all the elements are copied, and hence the time taken will be roughly proportional to the length of `w`.

You can shrink a vector by one element by writing `v.pop_back()`. But you can also set the size arbitrarily by writing

```
v.resize(newSize);
w.resize(newSize,newValue);
```

The first statement would merely change the size. The second statement would change the size, and if the new size is greater, then the new elements would be assigned the given value.

22.2.1 Inserting and Deleting Elements

It is possible to insert and delete elements from the middle of a vector. This is discussed in Section 22.5.2.

22.2.2 The Type `size_t`

We mentioned above that the member function `size` returns the size of the vector. This is true but it should be noted that the return type of `size` is `size_t`, which is an alias for `unsigned int`. The type name `size_t` is created in C++ to specially denote quantities that relate to sizes of objects in memory. Thus, the `size` member function on vectors as well as strings returns a value of type `size_t`. Note that the member function `find` in the `string` class discussed earlier also returns the index as a value of type `size_t`.

There are pitfalls in performing comparisons between signed and unsigned `int`s as noted in Section 6.8. To avoid these, often variables used to index into a vector, especially if they are compared to the vector size, are declared to be of type `size_t`.

22.2.3 Index Bounds Checking

Instead of using subscripts `[]` to access elements, you can use the member function `at`. This will first check if the index is in the range 0 (inclusive) to array size (exclusive). If the index is outside the range, the program will halt with an error message. Note that the `at` function can be used on the left as well as the right-hand side of assignments.

```
vector<int> v;
for(int i=0; i<10; i++) v.push_back(i*10);

v.at(0) = v.at(1);
```

This will cause the first element to be copied to the zeroth, i.e. at the end `v` will contain 10, 10, 20, 30, 40, 50, 60, 70, 80, 90.

22.2.4 Functions on Vectors

A vector can be passed to functions by value or by reference. Because a vector is a class, if passed by value the entire vector is copied, element by element. Thus the called function gets a new copy, and the the called function can make modifications only to the copy and not the original. However, when passed by reference, the called function gets access to the original and the values in the original may be read or modified.

Here are functions to read values into a vector and print values in the vector. We have considered vectors of `int` in this example.

```
void print(vector<int> v){
  for(size_t i=0; i<v.size(); i++) cout << v[i] <<' ';
  cout << endl;
}
void read(vector<int> &v){
  for(size_t i=0; i<v.size(); i++) cin >> v[i];
}

int main(){vector<int> v(5); read(v); print(v);}
```

We may of course templatize the functions, e.g.

```
template<class T>
void print(vector<T> v){
  for(size_t i=0; i<v.size(); i++) cout << v[i] <<' ';
  cout << endl;
}
```

It is usually good to pass vectors by reference, so that there is no unnecessary copying.

22.2.5 Vectors of User-defined Data Types

We can make vectors of objects of class `T`, so long as the class `T` has an assignment operator, a copy constructor and a destructor. This is because the vector class will call these member functions internally. So for example, you may write

```
vector<V3> v3vec;
vector<Circle> circles;
```

where `V3` is the class from Chapter 17, and `Circle` from Chapter 5. You can also make vectors of pointers.

```
vector<Circle*> circlevec;  // allowed.
```

22.2.6 Multidimensional Vectors

Since the template parameter in a vector names a type, by specifying that as a vector we can get a vector of vectors, i.e. equivalent of a two-dimensional array.

```
vector<vector<int> > v;
```

This simply defines `v` to be a zero-length vector of zero-length vectors. Notice the space between the two > characters. Without this space, the two > characters would be interpreted as the input extraction operator >>.

Here is how we might define a length 10 vector of length 20 vectors, i.e. a 10×20 matrix.

```
vector<vector<int> > w(10, vector<int>(20));
```

In this, we have used the two-argument constructor for constructing `w`. Its first argument, 10, specifies the length, and the second element, `vector<int>(20)`, gives the value of each element. But this value is itself a vector of length 20. Thus, we get a 10 by 20 matrix represented.

We can access the elements of the matrix in the usual manner, i.e. by writing `w[i][j]`. However, we may also modify whole rows if we wish. Thus, for `w` as defined above, we write:

```
w[0] = vector<int>(5);
```

we will change `w` to become a peculiar structure: it will have 10 rows; the first will have 5 elements, and the remaining will continue to have 20 elements.

This flexibility is very useful. Often in scientific computing, we encounter matrices of certain shapes, e.g. lower triangular matrices. In a lower triangular matrix, all elements above the main diagonal are 0. Thus, we need not even store them. So we can create a vector of vectors in which the `i`th vector (`i` starting at 0) has length `i+1`. This is an easy exercise.

On the one hand, the flexibility described above is useful, but on the other, creating a matrix as discussed above is also a bit verbose. Also, if someone uses a vector of vectors in a program, there is always the suspicion that they may be changing the sizes of the rows as described above. It is easier to understand a program if we are assured that a particular name always refers to a matrix 10×20 matrix, and that some function will not suddenly change it to become a 5×5 triangular matrix. In other words, we want to signal to the reader that we are really using only the usual kind of matrix operations, not using all the vector functions. For this, we can create a matrix class.

22.2.7 A Matrix Class

A *safe* matrix class is defined below. It does not allow the size of the individual rows to be changed once created, and only allows access to elements for reading and writing.

```
class matrix{
  vector<vector<double> > elements;
public:
  matrix(size_t m, size_t n) : elements(m, vector<double>(n)){}
  double &operator()(size_t i, size_t j){return elements[i][j];}
  size_t nrows(){return elements.size();}
  size_t ncols(){return elements[0].size();}
};
```

It can be used in a main program such as the following.

```
int main(){
  matrix D(10,10);          // 10 x 10 matrix

  for(size_t i=0; i<D.nrows(); i++){
    for(size_t j=0; j<D.ncols(); j++)
      D(i,j) = (i==j);      // access i,j th element
  }

  for(size_t i=0; i<D.nrows(); i++){
    for(size_t j=0; j<D.ncols(); j++)
      cout << D(i,j) <<' ';
    cout << endl;
  }
}
```

As you can see, we have overloaded the function call operator to access the elements. This is because we need to supply two indices, and the indexing operator `[]` can only take one index. Thus, the function call operator is more convenient. Also note that as defined, the default assignment operator is available to the class. You can disable that if you wish by making it private.

The class can be templatized so as to form a matrix of arbitrary type `T` rather than a matrix of type `double`.

22.3 SORTING A VECTOR

The standard template library contains many useful functions which you can access by including the header file `<algorithm>`.

If you include this file, sorting a vector `v` is easy. If the binary operator < is defined over the vector elements, you simply write

```
sort(v.begin(), v.end());
```

That's it! This function will sort the vector `v` in-place, i.e. the elements in `v` will be rearranged so that they appear in non-decreasing order as per the operator <. The arguments to the `sort` function indicate what portion of the vector to sort. By writing `v.begin()` you have indicated that the portion to sort starts at the beginning of `v`, and `v.end()` indicates that the portion to sort ends at the end of the vector. In other words, the entire vector is to be sorted. The expression `v.begin()` evaluates to an *iterator*. An iterator, which we will discuss in Section 22.5, is a generalization of pointers. The expression `v.end()` is also an iterator.

The sorting order can also be specified by supplying an extra argument which abstractly implements the < operation. The `sort` function then sorts in ascending order with respect to the specified extra argument. The extra argument can be given as

1. A lambda expression (Section 12.2)
2. A function pointer (Section 12.1)
3. A function object (Section 18.4)

Each of the above must take two arguments and return `true` if and only if the first argument should be considered smaller. We give examples of all these in Section 22.3.1.

Finally, Section 22.3.2 shows how the `sort` function from `<algorithm>` can also be used to sort arrays.

22.3.1 Example

We consider a variation of the marks display program of Section 14.2.2. Suppose we are given a file on each line of which appears the following information.

```
roll-number physics-marks math-marks chemistry-marks
```

Our goal is to read the file and then print out the information in 4 ways: sorted by roll number, sorted by physics marks, sorted by math marks, sorted by chemistry marks.

The natural way to write this program is to use a structure in which the information from a single line would be stored. Each such structure would be pushed back onto a vector which would then contain all the information read. Here is the structure we use.

```
struct student{
  int rollno;
  float physics, math, chemistry;
  bool operator<(const student& rhs) const{
                                    // used by the sort function
    return rollno < rhs.rollno;  // note the two const keywords
  }
};
```

To help in sorting by roll number, we have defined a comparison operator < which will compare roll numbers in order to decide which structure is smaller. Note an important point: the `sort` function requires that the function `operator<` be defined with both the receiver and the argument declared `const`.

In our main program, we will define a vector of `student` structures.

```
vector<student> svec;
```

We can read data into this by writing

```
student s;
while(cin >> s.rollno){
  cin >> s.physics >> s.math >> s.chemistry;
  svec.push_back(s);
}
```

Now to sort by roll number, we can simply write

```
sort(svec.begin(), svec.end());          // will use operator<
```

If we wanted to sort by another criterion, we could have redefined `operator<` appropriately. However, if we want to sort the same structure in more than one ways, then we must supply a < operator as an additional argument to the sort. This can be done in three ways, and below we show each.

To sort by physics marks, we will use a lambda expression.

```
sort(svec.begin(), svec.end(),
     [](const student& a, const student& b)
       { return a.physics < b.physics;});
```

As you can see, the lambda expression takes two arguments `a, b`, and says whether the marks in `a` are smaller than those in `b`. Using this, the `sort` function can decide how to reorder the vector.

To sort by math marks, we will use an external function

```
bool compareMarksFunction(const student& a, const student& b){
  return a.math < b.math;
}
```

which must be defined before the main program. In the main program, we call `sort` supplying this function (Section 12.1).

```
sort(svec.begin(), svec.end(), compareMarksFunction);
```

Finally, to sort by chemistry, we will use the fourth option at our disposal. We will supply the comparison function as a function object. Here is its definition, which you must put before the main program.

```
struct compareRollnoStruct{
  bool operator() (const student& a, const student& b){
    return a.chemistry < b.chemistry;
  }
};
```

We can use this to get sorting to happen as per the chemistry marks by writing

```
sort(svec.begin(), svec.end(), compareRollnoStruct());
```

In this, the third argument is a call to the constructor of the class `compareRollnoStruct`, i.e. an object of that class is constructed and supplied. The `sort` function can invoke it as a function to decide which of two objects is < the other.

As you can see, we have made four calls to `sort` in the above discussion, all use different ways of determining the final order. If we simply specify the start and end iterators, then we get sorting as per the < operator which must be defined for the objects that we are sorting. If we are sorting fundamental data types, the operator will be predefined; for user defined types, it is fine if we define it ourselves as above.

We also have a choice of specifying the operator as an extra argument to `sort`. The simplest way to do this is to use a lambda expression, which is what we did above for sorting in order of the physics marks. But we can use other ways too: specifying a non-member function, or specifying a function object. We used these also in the above code, to illustrate their use.

22.3.2 Sorting an Array

The algorithms in `<algorithm>` can also be used for sorting arrays. Here is an example.

```
#include <simplecpp>
#include <algorithm>
int main(){
  int a[5]={10,8,11,3,4};

  sort(a, a+5);
  for(int i=0; i<5; i++) cout << a[i] << endl;

  sort(a, a+5, [](int i, int j){return i>j;});
  for(int i=0; i<5; i++) cout << a[i] << endl;
}
```

The simplest way is to call `sort` with an iterator that points to the zeroth element of the array, and an iterator that points to an imaginary element that might lie past the last element in the array. We have remarked that iterators are generalized pointers; in case of arrays they are actual pointers! Thus, if `a` is the name of a 5-element array, then `a` points to the zeroth element and `a+5` points to an element which will lie past the last element (Section 15.1.5). Thus `sort(a, a+5)` in the above program will sort the array. The default is to sort in non-decreasing order. But you can change that using all the ideas discussed so far. The second *sort* command in the program above shows how you can get the array to be sorted in non-increasing order by supplying a comparison operator.

22.4 THE `map` TEMPLATE CLASS

The simplest way to think of the `map` class is as a generalization of an array or a `vector`. In an array or a vector, the index is required to be an integer between 0 and the $n-1$ if the length of the array is n. In a map, this condition is severely relaxed: you are allowed to use any value as the index, it need not even be numerical! As in an array, the value of the index determines which element of the map is being referred to.

To use the `map` template class, you need to include the header `<map>`. Next, you declare the map you want.

```
map<indexType,valueType> mapname;
```

This causes a map named `mapname` to be created. It stores elements of type `valueType`, which can be accessed by supplying indices of type `indexType`. It is required that the operator `operator<` be defined for the type `indexType`. Of course, if the operator is not originally defined, you can define it. However, the definition should have the usual properties expected of a comparison operator, i.e. it should be transitive and asymmetric.

Let us take a simple example. Suppose we want to store the population of different countries. Then we can create a `map` named `population`, which will store the population value (numeric). Say we store the population in billions as a unit, so our `valueType` is `double`. We would like to use the name of the country to access the element corresponding to each country, so our `indexType` could be string. So we can define our map as follows:

```
map<string,double> population;
```

The `string` class already has a < operator, so we do not need to do anything more. Next we insert the information we want into the map, i.e. we specify the population of different countries.

```
population["India"] = 1.21;  // population of India is 1.21 billion
population["China"] = 1.35;
population["Unites States"] = 0.31;
population["Indonesia"] = 0.24;
population["Brazil"] = 0.19;
```

The first line, for example, creates an element whose value is 1.21, and whose index is `"India"`. You use an array access like syntax also to refer to the created elements. For example, the following

```
cout << population["Indonesia"] << endl;
```

will print 0.24, which is the value stored in the element whose index is `"Indonesia"`.

You have to realize that while the statements look like array accesses superficially, their implementation will of course be very different. Effectively, what gets stored when you write `population["India"] = 1.21;` is the pair `("India",1.21)`. The name `population` really refers to a collection of such pairs. Subsequently, when we write `population["India"]` we are effectively saying: refer to the second element of the pair whose first element is `"India"`. So some code will have to execute to find this element (Section 22.4.2). So a lot is happening behind the scenes when you use maps.

What if you write two assignments for the same index, e.g.

```
population["India"] = 1.21;
population["India"] = 1.22;
```

This will have the effect you expect: the element created the first time around will be modified so that the value stored in it will change from 1.21 to 1.22.

An important operation you might want to perform on a map is to check if the map contains an element with a given index. Suppose you have read in the name of a country into a string variable `country`. Say you want to print out the population of that country if it is present in the map; else you want to print out a message saying that the population of that country is not known to the program. You can write this as follows:

```
cout << "Give the name of the country: ";
string country;
cin >> country;
if (population.count(country)>0)
  cout << population[country] << endl;
else cout << country << " not found.\n";
```

This code should follow the code given above for defining the map `population` and specifying the population of the various countries.

In this code, the member function `count` takes as argument an index value, and returns 1 if an element with that index is present in the given map. Thus, suppose the user typed in `"India"`, in response to our request to give the name of a country. Then `population.count(country)` would return 1 because we did enter the population of `"India"` into the map earlier. So in this case, the final value entered, 1.22, will get printed. On the other hand, if the country typed in was `"Britain"`, then `population.count(country)` would return 0, and hence the message "Britain not found." would be printed. Another way of determining whether a map contains a certain entry is discussed in Section 22.5.1.

You may wonder what would happen if we anyway execute

```
cout << population["Britain"] << endl;
```

without assigning a value to `population["Britain"]` earlier in the code. The execution of this statement is somewhat unintuitive. In general, suppose we have defined a map

```
map<X,Y> m;
```

and suppose `x` is a value of type `X`. Then if we access `m[x]` without first assigning it a value, then implicitly this first causes the statement `m[x]=Y();` to be executed, i.e. an element is created for the index `x`, and the element stores the value `Y()` obtained by calling the default constructor of class `Y`. After that the value of `m[x]` is returned. Thus in the case of the statement `cout <<population["Britain"] <<endl;`, the statement `population["Britain"]=double();` is first executed. The constructor for the type `double` unfortunately does not initialize the value. So the map will now contain an element of unknown value but having the index `"Britain"`. Hence, this unknown value would get printed.

22.4.1 Marks Display Again!

In this, we will have the teacher enter the names of the students instead of the roll numbers. We will consider the original problem, i.e. students walk up to the computer and want to know their marks. But this time, they type in their name rather than the roll number. Clearly, we can use `string`s to represent student names, and a `map` to store marks of students.

To make the problem more interesting, we will assume that for each student we have the marks in Mathematics, Physics, and Sanskrit. Further assume that the names are given in a file with lines such as the following.

```
A. A. Fair, 85, 95, 80
Vibhavari Shirurkar, 80, 90, 90
Nicolas Bourbaki, 99, 98, 75
```

i.e. the file will contain a line for each student with the name appearing first, succeeded by a comma, following which three numbers would respectively give the marks in the different subjects. The numbers are also separated by commas. This format, in which each line of the file contains values separated by commas, is often called the CSV format, or the "comma-separated values" format.

We will use a `string` to store the student name. To store the marks, we will use a structure.

```
struct Marks{
  double science, math, sanskrit;
};
```

The marks will be stored in a map, whose index will be the name of the student given as a string.

```
map<string,Marks> mark_map;
```

Say our file containing the marks is named `marks.txt`. Then we can declare it in our program as

```
ifstream infile("marks.txt");  // needs #include <fstream>
```

Next we discuss how to read values from a file in the CSV format. For this, we can use a form of `getline` function which allows a delimiter character to be given. The signature for this is:

```
istream& getline(istream& instream, string stringname, char delim)
```

In this, `instream` is the name of the input stream from which data is being read. The parameter `stringname` is the name of a string, and `delim` is the character that delimits the read. Thus, data is read from the stream `instream` until the character `delim` is found. The character `delim` is discarded, and the data read till then is stored into string `stringname`. Thus, we can read the name of a student by executing something like

```
string name;
getline(infile,name,',');
```

Used with the file above, this statement will cause `name` to get the value "A. A. Fair", including the spaces inside it. Subsequently, if we execute

```
getline(infile,name,',');
```

again, the string `name` would then hold the string `"85"`. Of course, we would like to convert this to a double, so we can use a stringstream (Appendix E).

```
double mmath;
stringstream(name) >> mmath;    // need #include <sstream>
```

This would cause the string `name` to be converted into a stringstream, from which we read into the variable `mmath`. Similarly, the other data can be read.

Figure 22.1 contains the entire program based on these ideas. In the first part, the file is read into the map `mark_map`. The first three values on each line, the name, the marks in math and the marks in science are comma separated. So they are used as discussed above. The last field is not comma separated, so it can be read directly. Note that when reading using the operator », the end-of-line character is not read. So before the next line is to be read, it must be discarded.

In the second part, the program repeatedly reads the names of students. If a name is present in the map, then the corresponding marks are printed.

22.4.2 Time to Access a Map

The (index,value) pairs constituting a map are stored using binary search trees (Section 24.1.1). As will be discussed in Section 24.1.6, making an access such as `population[country]` happens fairly fast, i.e. in time proportional to $\log_2 n$, where n is the number of countries for which data is stored in the map.

22.5 CONTAINERS AND ITERATORS

The classes `vector` and `map` are considered to be *container* classes, i.e. they are used to hold one or more elements. Even a `string` is thought of as a container because it contains sets of characters. There are other containers as well in the Standard Library, and we will glance at some of them shortly.

The Standard Library allows some generic processing of containers, be they vectors, or maps, or even strings. For this, it is necessary to be able to refer to the elements of the container in a uniform manner. This is accomplished using an `iterator`.

```
#include <simplecpp>
#include <fstream>
#include <sstream>
#include <map>

struct Marks{
  double science, math, sanskrit;
};

int main(){
  ifstream infile("students.txt");
  map<string,Marks> mark_map;

  Marks m;
  string name;

  while(getline(infile,name,',')){
    string s;
    getline(infile,s,',');
    stringstream (s) >> m.math;
    getline(infile,s,',');
    stringstream (s) >> m.science;
    infile >> m.sanskrit;   // read directly, not comma terminated
    getline(infile,s);      // discard the end of the line character

    mark_map[name] = m;     // store the structure into the map
  }

  while(getline(cin,name)){
    if(mark_map.count(name)>0)
      cout << mark_map[name].math << " " << mark_map[name].science
           << " " << mark_map[name].sanskrit  << endl;
    else
      cout << "Invalid name.\n";
  }
}
```

Fig. 22.1 *Program for another marks-display variation*

An `iterator` can be thought of as a generalized pointer to an element in a container. It is intended to be used in a manner analogous to the use of an (actual) pointer in the following code which applies a function `f` to all the elements of an array.

```
int A[10]
int* Aptr
for(Aptr = A; Aptr<A+10; Aptr++)  f(*Aptr);
```

In this code, we initialize the (actual) pointer `Aptr` to point to the zeroth element of `A`, and then increment it so that it points to successive elements. In each iteration, we dereference it and then apply the function `f` to it. Implicit in this code is the idea that the elements are ordered in a unique manner: specifically the elements are considered in the order in which they are stored in memory.

Now we see how we can write analogous code for containers. Analogous to the actual pointer `Aptr`, we will have an iterator which will abstractly point to elements of the container, and which we can step through as the execution proceeds. In general, an iterator for a `map` can be defined as follows.

```
map<X,Y> m;
map<X,Y>::iterator mi;
```

Here, `mi` is the iterator, and its type is `map<X,Y>::iterator`. Next we need to say how to set it to "point" to the first element in the map, and then how to step it through the elements. For this, we first need to fix an ordering of the elements stored in the container. For `vectors` and `maps`, the elements are considered ordered according to the index, i.e. the first element is the element with the smallest index. The member function `begin` on the container returns an iterator value that abstractly point to this first element. Thus, we can initialize our iterator by writing:

```
mi = m.begin();
```

An iterator supports two operations: by dereferencing you get to the element abstractly pointed to by the iterator, and by using the operator ++, the iterator can be made to point to the next element in the container (as per the < order). Finally, to determine when the iterations should stop we need to know when the iterator has been incremented beyond the last element in the container. For this, the member function `end` on the container is defined to abstractly point beyond the last element, just as the address `A+10` in the example above points beyond the last element of the array.

Suppose we wish to merely print all the elements in a container. Then here is how this can be done using iterators.

```
vector<float> mvec;
// code to insert elements into mvec
for(vector<float>::iterator mi = mvec.begin(); mi != mvec.end();
   ++mi)
  cout << *mi << endl;
```

The code for map containers is similar. When we dereference a map iterator, we get an element of the map, which is an (index,value) pair. The pair that we get is a (template) `struct`, with data members `first` and `second` which hold the index and the value respectively. Since we consider an iterator to be a pointer, the struct elements can be accessed using the operator ->. Here is how we can print out the map `population` of Section 22.4.

```
for(map<string,double>::iterator Pi = population.begin();
    Pi != population.end();
    ++Pi)
  cout << Pi->first <<": " << Pi->second << endl;
```

Similar code can be written for the `string` class. Note that the dereferencing operator * or the incrementation ++ should not be understood literally, these operators are given to you appropriately

overloaded. But you don't need to worry about all this; you can consider iterators to be abstractions of pointers for the purpose of using them.

22.5.1 Finding and Deleting `map` Elements

Iterators are specially important for the `map` class. We can use the `find` operation on iterators to get to an (abstract) pointer to an element which has a given index value. Thus, to see if the value `"Britain"` is stored in the map `population`, we can write

```
map<string,double>::iterator Pi = population.find("Britain");
```

If `"Britain"` is not present, then `Pi` would take the value `population.end()`. So to see if `"Britain"` is present and print its population we can write

```
map<string,double>::iterator Pi = population.find("Britain");
if(Pi != population.end())
  cout << Pi->first << " has population "<<Pi->second << endl;
```

You can delete the element pointed to by an iterator by using the `erase` function as follows.

```
map<string,double>::iterator Pi = population.find("Indonesia");
population.erase(Pi);
```

This would remove the entry for Indonesia.

22.5.2 Inserting and Deleting `vector` Elements

Iterators can be used with vectors for inserting and deleting elements. For example, we could write

```
vector<int> v;
for(int i=0; i<10; i++) v.push_back(i*10);

vector<int>::iterator vi = v.begin()+7;
v.insert(vi,100);                        // inserting into a vector

vi = v.begin() + 5;
v.erase(vi);                             // deleting an element
```

The first two statements respectively declare a vector `v` and set it to contain the elements 0, 10, 20, 30, 40, 50, 60, 70, 80, 90. The third statement causes `vi` to point to the seventh element of `v`, i.e. the element containing 70. Then 100 is inserted at that position, the elements in the positions seventh onwards being moved down one position. The size of the vector of course increases by one. After that we set `vi` to point to the fifth element. Then that element is deleted. This causes the subsequent elements to be moved up one position. Thus, at the end the vector `v` would contain the elements 0, 10, 20, 30, 40, 60, 100, 70, 80, 90.

22.6 OTHER CONTAINERS IN THE STANDARD LIBRARY

The Standard Library has several other containers which are very useful.

For example, the container `deque` is a double-ended queue, which supports element insertion and removal, from the front as well as the back. The container `queue` allows insertions at the back and

removal from the front, while the container stack requires that insertions and removals both be done from the same end.

An important container is the priority queue. You can insert elements arbitrarily, however, when removing elements, you always get the *smallest* element inserted till then. We discuss and use priority queues in Chapter 27.

An interesting container is the `set`. This supports operations for inserting elements and subsequently finding them. The elements are required to have `operator<` defined on them, and this order is used for storing the elements in a binary search tree (Section 24.1.1). It turns out that finding an element in a set or inserting an element in a set both take time proportional to $\log n$ where n is the number of elements present in the set at the time of the operation. We will explore the reasons behind this in Section 24.1.

These descriptions are very brief. You should consult various Standard-Library references on the Web to get details.

22.7 THE `typedef` STATEMENT

The `typedef` statement can be used to create a new name for an existing type.

```
typedef existingType newName;
```

So for example, you can write

```
typedef map<string,double> popType;
typedef vector<vector<double> > matrix;
```

So with these definitions, you can write `popType` and `matrix` instead of the longer names, and save yourself typing and perhaps make your programs more readable.

Of course, the `typedef` statement is not in any way limited to being used with container types from the standard library. It can be used also for ordinary types.

```
typedef double mynum;
```

With this, you could use `mynum` as a synonym for `double`. This is useful in case you decide one day that you really want to represent the numbers in your program using `long double`. If you had declared them to be of type `mynum`, then you would only need to make the change in the definition of `mynum`, rather than change the definition of every numerical variable in your program.

22.7.1 More General Form

The above type definitions could be considered to be only convenient, but not providing new capability. This is because you could textually substitute `existingType` for `newTypename`. However, there is a general form which actually provides new capability. The form is

```
typedef existingType newTypeExpression;
```

This defines an equivalence between `existingType` and `newTypeExpression`. Here is an example.

```
typedef double (*fptrtype)(double);
```

This says that `double` is the same as the type you get when you dereference something of type `fptrtype`, and then apply that to arguments of type `double`. In other words, `fptrtype` is of type pointer to function that takes a `double` as argument and returns a `double`. As you can see, there is no other way to define the type `fptrtype`. With this definition of `fptrtype`, you could define `pf` from the end of Section 12.1 as follows.

```
fptrtype pf;
```

22.8 REMARKS

You have probably guessed by now that the classes we discussed in this chapter would have to be implemented in the style of the `String` class discussed in Chapter 21. Indeed, that is true. They will use heap memory to store data, and allocate and deallocate heap memory when needed.

The important point to note however is that you don't have to worry about the implementation in order to use these classes. Indeed, the constructors, destructors, copy constructors, assignment operators of these classes have already been written, so that there are no memory leaks, dangling pointers, etc. You don't need to worry about memory allocation; indeed you should be able to do everything you want without ever having to use the `new` and `delete` operators.

EXERCISES

1. Explain what each statement of the following code fragment does.

```
vector<int> a(5,33);
vector<char*> countries(4);
vector<vector<double> > v(3,vector<double>(5, 3.14));
```

2. Write a code fragment that creates a 10×10 matrix stored as vector of vectors of doubles and initializes it to the identity matrix.
3. Write a program to multiply two matrices of arbitrary sizes represented as vector of vectors.
4. Write a function which returns a lower triangular matrix using a vector of vectors. Specifically, you should only allocate space to store elements a_{ij} where $j \leq i$.
5. Define a class `LTM` for storing lower triangular matrices, i.e. matrices in which elements with indices `(i,j)` are 0 if `i<j`. It should have a signature as follows.

```
class LTM{
  vector<vector<double> > data;
public:
  LTM(int n);
  double getElem(int i, int j);
  void setElem(int i, int j, double v);
}
```

As you might guess, the constructor constructs an LTM matrix with the given number of rows and columns. The member functions return the element at index `i,j` and assign the value `v` to the element at index `i,j` respectively. Note that if `j>i` then `getElem` must return 0. If `j>i`

the `setElem` must do nothing and print a message saying that the operation is not allowed. Give implementations of all the member functions.

6. Write a program that will receive information about the states of India and their capitals and answer questions about these when asked. Specifically, it should process 3 kinds of commands. The first kind is

```
Learn state capital
```

As an example, the user may type `Learn Maharashtra Mumbai`. In this case this information must be remembered by the program. The second kind of command is

```
Tell capital-or-state-name
```

For example, the user may type `Tell Gandhinagar`, whereupon the program must respond that it is the capital of Gujarat. Likewise, if the state is given its capital must be given in response. The third kind of command is just

```
Exit
```

whereupon the program must exit.

7. Write a program that prints out all positions of the occurrences of one string `pattern` inside another string `text`. Use appropriate functions from the `string` class.

8. Design a class to efficiently store sparse polynomials i.e. polynomials in which even if the degree is n, there may be far fewer terms in the polynomial, i.e. many of the powers might have coefficient 0. In such a case, it may be wasteful to allocate an array or vector of size $n+1$ to store a polynomial. Instead, it might be more efficient to store only the non-zero coefficients, i.e. store the pair (i, a_i) if the coefficient a_i of x^i is non-zero. Use a `map` to store such pairs. Write functions to add and multiply polynomials. Note that iterators on maps will go through stored pairs in lexicographical order. Exploit this order to get efficient implementations.

9. Suppose for each student we know the marks in several subjects. The total number of subjects might be very large, of which each student might have studied and got marks in some. Write a program which reads in the marks a student has obtained in different subjects, and then prints out the marks obtained given the name of a student and the name of the subject for which the marks are requested.

You are expected to use a `map` to store the data for all students, and a map for each student in which to store the marks for the different subjects taken by the student.

10. The algorithm collection in the standard library also contains a `binary_search` function for performing binary search on sorted containers such as vectors. The signature of this function is

```
bool binary_search(ForwardIterator first, forwardIterator last,
                   const T& value_to_search);
```

Here, the region of the container between `first` (inclusive) and `last` (exclusive) is searched to find an element equal to `value_to_search`. The type of the element stored in the container must be `T`. Use this to implement variation of the marks display program of Section 22.4.1, using just vectors rather than `maps`.

The function `binary_search` is guaranteed to execute in logarithmic time when used with `vector` containers.

11. Write a program which implements a dictionary of the English language. A natural representation would be a map, with the words being the indices and the meanings being the values. This will be suitable for exact look-ups. However, suppose we wish to find approximate matches too. This is because we will typically only store the *root* words in the dictionary, e.g. the word "dictionary", but not the words obtained from the root by inflection, e.g. the word "dictionaries". In such cases, when you look up "dictionaries", you would like to get the word that has the longest prefix match, which will likely be "dictionary" in this case. After that your program could decide whether the word you found can indeed be inflected to give the word you are looking for. For such processing, perhaps a simple (sorted) vector might be a better representation than a map. In any case, write a program which not only tells you whether the given word is in the dictionary, but also whether it is likely an inflection of a word in the dictionary.

CHAPTER 23

Representing Networks of Entities

Many real-life systems can be considered to be collections of entities which are somehow linked together into a network. For example, a circuit consists of components connected together by wires. Roads connect cities. When you browse the internet, you can go from one page to another by clicking on a link; a link thus connects one page to other pages. Or the entities might be people, where each person could be considered linked (attached!) to his/her friends. This chapter will give an introduction to computations relating to such networks.

There may be several questions we could ask about such networks. For a circuit, we might want to know the currents and voltages in the different components. In a road map we might want to know the shortest path to go from one city to another. We might want to determine the importance of each page on the Internet. Search engines have to routinely answer this question when they have to show results of a Web search—the more important pages must be listed before the less important ones. In a network of friends, you might perhaps want to know who has the largest number of friends, or whether two individuals have a mutual friend. Many such questions are hard to answer and require sophisticated mathematical and computer science ideas.

The basic mathematical model used to represent networks is a *graph*, which we discuss in Section 23.1. On a computer there are two common representations for graphs. In Section 23.2, we consider the *adjacency list* representation. A more detailed example of the adjacency list representation appears in Section 27.4. The representation for graphs in Chapter 24 is also a variation on adjacency lists. In Section 23.3, we discuss the *adjacency matrix* representation. Adjacency matrix like representations are used very commonly, especially in engineering. We discuss two such applications: analysis of an electrical circuit, and analysis of the graph consisting of the pages on the internet and the links between them.

23.1 GRAPHS

A graph, as you might know, consists of two sets, V, a set of *vertices*, and E, a set of *edges*. Each edge is a pair (u, v), where $u, v \in E$. An edge (u, v) is said to connect the vertices u, v, or make them *adjacent*. The edges may be *directed* or *undirected*, correspondingly, we may consider the pairs (u, v)

to be ordered or unordered. It is customary to say that an edge (u, v) is *incident* on vertices u, v. It is also common to use the term *node* instead of the term vertex.

As you might guess, vertices correspond to the entities in a network, and edges to the connections/relationships between them. We can associate *attributes* with vertices and edges. For example, each vertex may have a name attribute, corresponding to the name of the associated entity. Each edge may be associated with a number called its *weight*, which indicates say, the strength of the connection between the entities. Undirected edges are used to represent symmetric relationships, e.g. friendship. Directed edges represent asymmetric relationships, e.g. X is a *follower* of Y. As another example: in a road network, we will typically represent one way roads by a directed edge, whereas a road allowing bidirectional travel might be represented by an undirected edge or by two directed edges, one in each direction.

Most commonly, vertices are represented by objects in C++. Often, a network contains entities of only a single type, in which case the corresponding vertices will be represented by objects of a single class. But it is possible to have different types of entities. For example, you may have a network in which the entities are authors and books, with links between books and their authors. For this, you will have vertices of class author and also vertices of class book in your graph.

Edges may also be represented by objects, or they may be represented more simply.

23.2 ADJACENCY LISTS

In the simplest case, an edge is represented using a pointer. Thus, if there is a directed edge from a vertex u to a vertex v, the object corresponding to u will contain a pointer to the object corresponding to v. If an edge (u, v) is undirected, then we will have a pointer to v in object u, as well as a pointer to u in object v. In general, each vertex will have many edges. In this case the associated pointers will be stored in a list, hence the name adjacency list. In C++, it is most natural to use `vectors` to build lists.

As an example, we show how a network of friends may be represented. First, we need a `struct` to represent each person.

```
struct Person{
  string name;
  vector<Person*> friends;
};
```

Ideally, the data members should be private, accessed using suitable member functions. We have made data members public here for the sake of simplicity.

The following code creates five persons and fills in their names.

```
  Person persons[5];

  persons[0].name = "Harry";
  persons[1].name = "Hermione";
  persons[2].name = "Ron";
  persons[3].name = "Draco";
  persons[4].name = "Crabbe";
```

To make Harry and Hermione friends of each other, we merely have to add an undirected edge. As we have said, an undirected edge corresponds to pointers between the corresponding entities. Thus, we

must add a pointer to `persons[1]` in `persons[0].friends`, and vice versa. For this we will write a function.

```
void makefriends(Person &p, Person &q){
        // add an undirected edge in the graph
  p.friends.push_back(&q);
  q.friends.push_back(&p);
}
```

This can be called to add the required friendship edge, and others too.

```
  makefriends(persons[0], persons[1]); // Harry, Hermione
  makefriends(persons[2], persons[1]); // Ron, Hermione
  makefriends(persons[0], persons[2]); // Harry, Ron
  makefriends(persons[3], persons[4]); // Draco, Crabbe
```

Now if we want to print the friends of Hermione (stored in `persons[1]`), we merely write

```
  for(size_t i=0; i < persons[1].friends.size(); i++)
    cout << persons[1].friends[i]->name << endl;
```

23.2.1 Edges of Different Types

Each person need not have links only to his/her friends. Suppose we also want to have links to enemies. In that case, we merely add another data member `enemies` of type `vector<Person*>` to the `Person` class. Suppose that some of the persons have a unique favourite friend. Representing this is even simpler. Since we know that there is at most one favourite, we just add a data member `favourite` of type `Person*`. Thus, our definition now becomes

```
struct Person{
  string name;
  vector<Person*> friends, enemies;
  Person* favourite;
}
```

Given our preceding definitions, we can make Ron be Harry's favourite friend by writing

```
persons[0].favourite = &persons[2];
```

On the other hand, if we wanted to indicate that Harry has no favourite, we write

```
persons[0].favourite = NULL:
```

23.2.2 Array/Vector Indices Rather than Pointers

If we know that all the entities in our network will belong to a single array, e.g. the array `persons` as above, then an alternate representation is possible. Instead of storing a pointer to an object, we can store the index of the object in the vector. Thus, our definition of person would change as

```
struct Person{
  string name;
  vector<int> friends;
};
```

Assuming `persons` is a 5-element array initialized with names as given earlier, we can add an edge between Harry and Hermione as follows:

```
persons[0].friends.push_back(1);   persons[1].friends.push_back(0);
```

And the names of friends of `persons[1]` could be printed as

```
for(size_t i=0; i < persons[1].friends.size(); i++)
  cout << persons[persons[1].friends[i]].name << endl;
```

23.2.3 Edges Represented Explicitly

Suppose we wish to represent the network of roads between cities in a country. The cities will be the vertices in this network, and the roads themselves will be the edges. Since intercity roads are typically two way, we will use two edges for each road, one in each direction. We could make roads be pointers from one city to another; but we might also want to store the names of roads, and their lengths. So it is convenient to have objects to represent roads too.

A city object will store the name of the city and pointers to road objects representing the roads it is connected to. A road object will store the pointers to the city it leads to; optionally also the city it starts from, and say its name and length.

```
struct Road; // forward declaration so that we can write Road* below.
struct City{
  vector<Road*> roads;
};

struct Road{
  string name;
  City* from, to;
  double length;
};
```

We will see this idea developed in Section 27.4.

23.3 ADJACENCY-MATRIX REPRESENTATION

The edges of a graph can also be represented using a so called *adjacency matrix*. If the graph has n vertices, an $n \times n$ matrix A is used, with entry A_{ij} giving information about the edge from vertex i to vertex j, if any. For example, we might set $A_{ij} = 1$ to indicate that an edge is present, and $A_{ij} = 0$ to indicate that there is no edge. Other values can also be used, depending upon the context, as we will see later.

The main drawback of the adjacency matrix representation is the large memory required. An adjacency matrix has n^2 elements, and thus that much memory is used, no matter how many edges are actually present. If we use an adjacency list representation, then in each vertex object we will use just the memory needed to represent the edges incident on that vertex. Thus, the total memory used for edge representation is proportional to the number of edges, which could be far fewer than n^2. If a graph has only a few edges, then the adjacency list representation saves on memory.

However, there are advantages too for adjacency matrices. The most obvious advantage is that we can very easily check whether an edge from vertex i to vertex j exists: we simply examine the value

of A_{ij}. Also, as we will see shortly, in many applications, the adjacency matrix can be directly used in operations such as matrix multiplication, solving a system of equations, and so on. So in such applications, the adjacency matrix representation is very convenient.

The matrix stores the information about the edges. To store information about the vertices, we will need to use an additional vector.

As an example, we will show how to build the same friendship network as discussed in the previous section. We will write this as a class.

```
const int nFriends=5;
struct Person{
  string name;
};
class Fgraph{                              // Friends' graph
  vector<Person> vertices(nFriends);
  vector<vector<int> >  edges(nFriends, vector<int>(nFriends, 0));
public:
  setName(int i, string val){
    vertices[i].name = val;
  }
  makeFriends(int i, int j){
    edges[i][j] = edges[j][i] = 1;
  }
  printFriends(int i){
    for(int j=0; j<nFriends; j++)
      if(edges[i][j] == 1) cout << vertices[j].name << endl;
  }
};
```

The member `vertices` in the class `Fgraph` will store information about the vertices in the graph. The vertices represent friends, so we have made `vertices` a `vector` of `Person` objects. The object `vertices[i].name` will store the name of the `i`th person in the set. The member `edges` in `Fgraph` will store information about the edges. As you can see, it consists of a matrix of `int`s. We will set `edges[i][j]` to 1 if the person stored at `vertices[i]` is a friend of the person stored at `vertices[j]`. The member functions implement these ideas.

Now we can write a main program as follows.

```
int main(){
   Fgraph F;
   F.setNames(0,"Harry");
   ...
   F.makeFriends(0,1); // makes Harry and Hermione friends
   ...
   F.printFriends(2);  // prints the friends of Ron.
}
```

Adjacency matrices or their minor variations are used in many applications. We discuss two next.

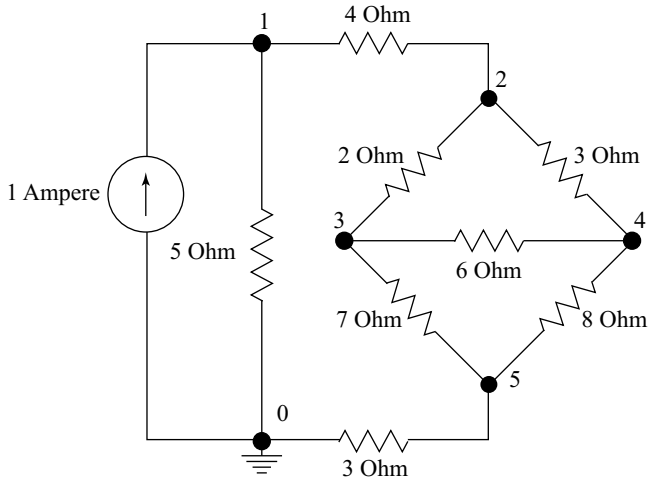


Fig. 23.1 *Circuit to be analyzed*

23.4 CIRCUITS

We will consider the problem of finding the currents and voltages at different points in a circuit consisting of resistors and current sources, such as the one shown in Figure 23.1. You may perhaps not be familiar with current sources. A current source is a circuit element out of which a specified amount of current flows no matter what the element is connected to. Circuits which contain voltage sources can also be analyzed, but the algebra is slightly more complicated. We will discuss this later.

First, we view this circuit as a graph, i.e. identify the vertices and edges in it. The resistances and the current source can be considered to be the edges, the points at which these attach to each other can be considered to be the vertices. There are 6 vertices in Figure 23.1, numbered 0 to 5, shown as solid circles. There are 9 edges, one consisting of the voltage source, and 8 consisting of resistors. It is possible to have circuits in which there are devices which have more than two electrical connections to them, such as transistors. The model for such graphs will be different.

A circuit is commonly represented by its so called conductance matrix, which is similar in spirit to the adjacency matrix of its graph. It is customary to name the matrix G. Indeed elements G_{ij} depend upon what is connected between vertices i, j in the network. Specifically, we will have

1. $\mathbf{G}_{ij} = -1/R_{ij}$: If R_{ij} is the value of the resistance connecting vertices i and j, where $i \neq j$. Note that if no resistor is shown between vertices i, j then we consider the resistance between them to be ∞, in which case $G_{ij} = 0$. You may know that a resistance of value R is the same as a conductance of value $1/R$. Thus, G_{ij} can be considered to be the negative of the conductance connected between nodes i, j.
2. $\mathbf{G}_{ii} = \sum_j 1/R_{ij}$: The value of G_{ii} is thus set to be the sum of the conductances connected to vertex i.

Note that the current source values does not figure in setting the values of the conductance matrix.

You can see that the conductance matrix can roughly be considered to be an adjacency matrix for the circuit: the entries are 0 when there is no connection, and the entries are large (though negative) when there is a high conductance between two vertices. But the real motivation in setting the entries in this manner is that it helps in circuit analysis.

Let V_i denote the voltage at node i. The goal of circuit analysis is to solve for the variables V_i. Considering V_i to be the elements of a vector V we can write down the following equation.

$$GV = S$$

Here, S is a column vector also having n elements, and S_i denotes the sum of the current leaving vertex i through the current sources attached to vertex i, if any. For our circuit, the equation with the matrix and vectors shown in full becomes

$$\begin{pmatrix} \frac{1}{5}+\frac{1}{3} & -\frac{1}{5} & 0 & 0 & 0 & -\frac{1}{3} \\ -\frac{1}{5} & \frac{1}{5}+\frac{1}{4} & -\frac{1}{4} & 0 & 0 & 0 \\ 0 & -\frac{1}{4} & \frac{1}{4}+\frac{1}{2}+\frac{1}{3} & -\frac{1}{2} & -\frac{1}{3} & 0 \\ 0 & 0 & -\frac{1}{2} & \frac{1}{2}+\frac{1}{6}+\frac{1}{7} & -\frac{1}{6} & -\frac{1}{7} \\ 0 & 0 & -\frac{1}{3} & -\frac{1}{6} & \frac{1}{3}+\frac{1}{6}+\frac{1}{8} & -\frac{1}{8} \\ -\frac{1}{3} & 0 & 0 & -\frac{1}{7} & -\frac{1}{8} & \frac{1}{3}+\frac{1}{7}+\frac{1}{8} \end{pmatrix} \begin{pmatrix} V_0 \\ V_1 \\ V_2 \\ V_3 \\ V_4 \\ V_5 \end{pmatrix} = \begin{pmatrix} 1 \\ -1 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

Note that in our example, $S_0 = 1$ because unit current must flow out of vertex 0 into the current source. Similarly, $S_1 = -1$ because unit current must flow in to vertex 1 from the current source. There are no current sources attached to the other vertices, and hence, the corresponding S_i are 0.

A matrix equation really consists of as many ordinary equations as there are rows. So for example, considering the first row of the product, we have

$$V_0(1/5 + 1/3) + V_1(-1/5) + V_5(-1/3) = -1$$

This equation really turns out to be saying that the total current entering vertex 0, through voltage sources and resistors, must equal 0, since charge cannot be created nor destroyed. This can be seen if we rewrite the equation a bit.

$$(V_0 - V_1) \cdot 1/5 + (V_0 - V_5) \cdot 1/3 + 1 = 0$$

In this you can see the first term $(V_0 - V_1) \cdot 1/5$ as the current flowing through the 5-ohm resistance. This current enters vertex 0. Similarly, $(V_0 - V_5) \cdot 1/3$ is the current entering through the 3-ohm resistance. Finally, 1 ampere is the current entering through the current source. These must add to zero, because of charge conservation. Indeed you will see the ith row of the matrix equation to simply be asserting that the total current leaving and entering vertex i must be 0. Thus, the matrix equation is valid.

Thus, analysis of this circuit is merely solving this set of equations! And we know how to do it from Section 15.2.1! There is one slight twist. The equations we have written down are not independent, i.e. they contain some redundant information. To see this, pick any equation. You will see that it can be obtained by adding up the remaining equations and multiplying the resulting equation by -1. Thus, we can throw out any equation and not lose information. You may wonder if we can do this one more time. Turns out that the answer is no—the remaining equations are independent, i.e. no equation in them can be obtained by combining other equations in any manner.[1] So now we seem to have run into a problem: we have n unknowns $(V_0, \ldots, V_{n-1})$ but only $n - 1$ equations. However, this is not a problem. As you may know, voltages in a circuit are relative, i.e. we can arbitrarily designate one of the voltages to be

[1] This assumes that our circuit does not have disconnected subcircuits.

0. For example, we can substitute $V_0 = 0$ into the matrix. Simultaneously, we throw out equation 0. Thus, we are left with $n-1$ equations in $n-1$ unknowns, which we can solve as per Section 15.2.1.

We can compute the currents in the different resistors by multiplying by the voltage drop across the resistance. Thus, the current through the 4-ohm resistance in the direction vertex 1 to vertex 2 is $(V_2 - V_1) \cdot 1/4$, which can be calculated given values of V_1, V_2.

We finally consider how to deal with voltage sources. Suppose we have a voltage source connecting vertices i, j. Then we will need to have an additional variable I_{ij} to represent the current through that voltage source. But we will also have an equation, viz. $V_i - V_j$ must equal the specified voltage source value. Thus, again we will have as many equations as there are unknowns, and we can solve the system.

23.5 SURFING ON THE INTERNET

When you give a query to a search engine, it must go through the all pages on the web and present to you pages that are relevant to your query. But in addition, you would also like the pages to be authoritative. How to identify authoritative pages is the question we consider here.

Here is a possible definition: the authoritativeness of a page equals the number of pages linking to it. The motivation behind this definition is that page A links to page B only because the creators of page A consider page B to be useful/trustworthy. In some sense, page A is recommending page B. Some additional observations could also be made. A recommendation can be considered to be more significant if we somehow decide that recommending page is itself authoritative. Thus, when page A links to page B, then it contributes to the authoritativeness of B, and to a first approximation we could say that this contribution is in proportion to its own authoritativeness. Next, suppose a page C links to 100 pages, whereas page page D links to only 4. It would seem that we should take the recommendations of D more seriously because it is perhaps being more selective in its recommendations. This suggests that we should consider that if a page links to another, then it contributes authoritativensss in direct proportion to its own authoritativeness and (to a first approximation) in inverse proportion to the number of links going out of itself.

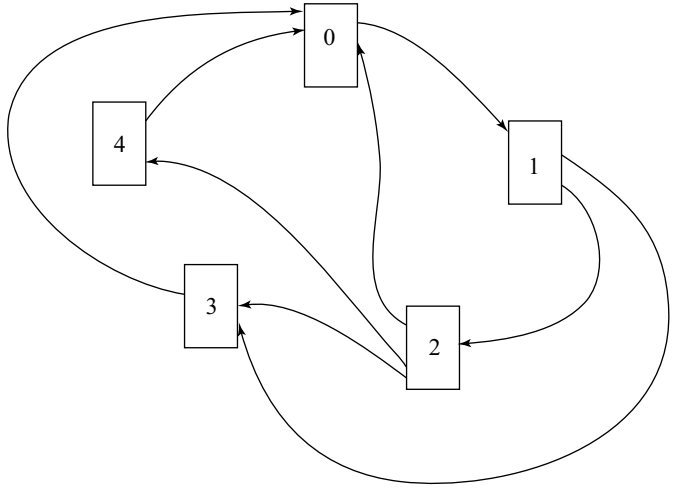


Fig. 23.2 *Pages with links*

So suppose x_i denotes the authoritativeness of a page i. Then the above discussion suggests that perhaps we could demand that

$$\text{For each page } i: \quad x_i = \sum_{j|j \text{ links to } i} \frac{x_j}{d_j} \tag{23.1}$$

In this, d_j is the number of outgoing links out of page j. Consider as an example, the network shown in Figure 23.2. In this, page 3 for example has links coming in from page 1 and page 2. Page 1 links to two pages; thus it contributes half its authoritativeness to page 3. Likewise page 2 links to 3 pages and so contributes a third of its authoritativeness. Thus, we would write for page 3:

$$x_3 = \frac{x_1}{2} + \frac{x_2}{3}$$

In general, for n pages, we will get n equations in the n unknowns $x_0, \ldots, x_{n-1}$. Letting x denote the vector consisting of elements $x_0, \ldots, x_{n-1}$, the above equations can be written in the form

$$Ax = 0$$

where A is a matrix having elements $A_{ii} = -1$ for all i, and $A_{ij} = \frac{1}{d_j}$. Thus, for the pages in Figure 23.2, we will get

$$\begin{pmatrix} -1 & 0 & \frac{1}{3} & 1 & 1 \\ 1 & -1 & 0 & 0 & 0 \\ 0 & \frac{1}{2} & -1 & 0 & 0 \\ 0 & \frac{1}{2} & \frac{1}{3} & -1 & 0 \\ 0 & 0 & \frac{1}{3} & 0 & -1 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} \tag{23.2}$$

These equations, as it turns out, has the same problem as the circuit equations: any single equation can be obtained by adding up the remaining equations and multiplying by -1. Thus we must throw out one of the equations. Now we seem to be left with just $n - 1$ equations in n unknowns. Thus, we will not get a unique solution.

Notice however, that Eq. (23.1) defines authoritativeness only relatively. Put another way, if a certain set $x_0, \ldots, x_{n-1}$ of authoritativeness values satisfy Eq. (23.1), the equation would also be satisfied by $2x_0, \ldots, 2x_{n-1}$. So we could fix one of the values, say x_0 to be 1. A more common strategy is to fix the sum to be 1. Thus, we will have an additional equation $x_0 + x_1 + \ldots + x_{n-1} = 1$. Now we have n equations in n unknowns which can be solved.

Thus, for the system of Figure 23.2, say we drop the first equation from equation (23.2), and include the equation $x_0 + x_1 + x_2 + x_3 + x_4 = 1$. If we solve this, we will get

$$x = (0.3, 0.3, 0.15, 0.2, 0.05)$$

This gives highest authority to pages 0 and 1, and very little authority to page 4. The high score for page 0 can be explained say because it has the largest number, 3, of links coming in. Page 1 has only one link coming, in, but that comes from a highly ranked page, page 0.

This is a simplified description of the *PageRank* algorithm proposed 15-20 years ago by the founders of the search engine Google. Today's search engines use additional, more sophisticated algorithms.

23.6 CONCLUDING REMARKS

We have described some ways of representing networks. However, there can be others. Especially if the network has some special structure, then we may be able to assign numbers to different vertices such that from the number of a node it is clear what other nodes are connected to it. We will see an example of this in Chapter 28, where we will suitably number the runways and taxiways so that it is clear from the numbers what other runways/taxiways a given runway connects to.

EXERCISES

1. Write a program that constructs representations about friendship and enemity as discussed in Section 23.2. It should take as input a file containing information about how many persons are there, who are the friends of each person, who are their enemies, and who are the favourite friends of each person if any.
   Write functions to (a) find whether two given persons have common friends, and (b) find the most popular person, i.e. the person who is named as the favourite by the largest number of persons.
2. We have noted that in many natural matrices, e.g. a conductance matrix, many elements are 0. To save memory, it is desirable to have a representation in which we only store the non-zero elements. Of course, the representation should be capable of listing out all elements of the matrix if needed (whether they are zero or non-zero), but it should not explicitly store the elements with value 0. Instead, if an element is not explicitly stored, it should be considered to be 0. Device such a representation. *Hint*: It will effectively be the adjacency list representation of the matrix.
   Write a member function for the class so that you can perform matrix vector multiplication using that representation.
3. Suppose in Figure 23.1 we have a voltage source of value 1 volt, between vertices 3 and 4 (with vertex 3 connected to the positive end of the voltage source). Add the relevant equation into your program and find the resulting currents.
   In general, extend your program to handle voltage sources.
4. Devise graphical editors to input each of the networks discussed in this chapter. For example to create a friendship network, your editor will have a button to create a person. Then additional buttons to create friendship links and so on. The editors should also have buttons which when clicked will suitably process the given network. Where relevant, device ways to show the result also on the graphics canvas.
5. Consider electrical circuits in which you have inductors. An inductor is associated with an inductance L. If a voltage V is applied across the terminals of an inductor for a very small time Δ, then the current through the inductor increases by $\frac{V}{L\Delta}$. Thus you can think of an inductor as a variable current source. Specifically, you must know the current I through the inductor at time 0; you can assume that at time 0 the inductor behaves like a current source of value I. At time Δ, it behaves like a current source of increased value, i.e. $I + \frac{V}{L\Delta}$. Use this idea together with ideas from Chapter 19 to determine how the voltages and currents in a circuit change if it contains inductors in addition to resistances and current sources. You can use either the first-order Euler method or the leapfrog method.

CHAPTER 24

Structural Recursion

We will begin this chapter with the following abstract problem: how to represent a set. For definiteness, suppose that our program only needs to perform two kinds of operations on a set. We may wish to *insert* an element into a set, and we may wish to *query* whether a certain integer is already in the set. Can we find a representation which allows both the operations: insertion and querying, to happen fast? Such representations are very useful, and are indeed used in the standard library containers `set` and `map`. In this chapter, we will study the main idea behind such representations.

The second problem in the chapter concerns the layout of mathematical formulae. Consider the following mathematical formulae:

$$\pi = \cfrac{4}{1+\cfrac{1^2}{3+\cfrac{2^2}{5+\cfrac{3^2}{7+\cfrac{4^2}{9+\ddots}}}}}$$

and

$$\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$$

Both are correct, and the first one is rather elegant. Our concern in this chapter, however, is not the validity or elegance of these formulae. Our concern is much more mundane: how do we layout these formulae on paper. Where do we place the numerator and the denominator, how long do we make the lines denoting division? What if the denominator is itself a complicated expression as is the case in the formula ("continued fraction expansion") for π? Can we have a computer program make all these decisions for us? While this is somewhat tricky, many programs are indeed available for doing this. The most important amongst them is perhaps the TEX program developed by Donald Knuth. The program TEX has a language for specifying mathematical formulae, and in this language, the two formulae above can be specified as

```
\pi = \cfrac{4}{1+\cfrac{1^2}{3+\cfrac{2^2}{5+\cfrac{3^2}
{7+\cfrac{4^2} {9+\ddots}}}}}
```

and

```
\sum_{i=1}^ni^2=\frac{n(n+1)(2n+1)}{6}
```

Given this textual description as input, TEX can generate layouts like the ones shown. While the textual description is quite cryptic, you can probably make some sense of it. You can guess, perhaps, that the symbol ^ is used by TEX to denote exponentiation. Or that `\frac` and `\cfrac` somehow denote fractions. Even without precisely understanding the language of TEX you can see that the input given to TEX does not contain any geometric information. The input does not say, for example, how long the lines in the different fractions need to be drawn. Indeed, all this is determined by TEX, using a nice blend of science and art.

Both problems: designing representations for sets and laying out mathematical formulae will involve entities which are defined in a recursive manner! It is customary to say that these entities have a recursive structure. Such entities are very useful, and this chapter will give a brief introduction to representing and processing (recursively, of course!) such entities.

24.1 MAINTAINING AN ORDERED SET

The simplest way to store a set is to use a `vector` (Chapter 22). To insert an element, we simply use the `push_back` function. To determine if an element is present, we can scan through the vector. The scanning operation, however, is rather time consuming: potentially we will examine every element stored in the vector. A slight improvement is to keep the elements sorted in the vector. Then we will be able to perform membership queries using binary search (Section 16.1), which would go very fast. However, when a new element is to be inserted, we will need to find its position, and shift down the elements larger than it. This operation will on the average require us to shift half the elements, and thus is quite time consuming. So again, this is unsatisfactory.

24.1.1 A Search Tree

There is a way to organize the elements of the set so that insertions as well as membership queries can be done very fast: we store them in a so called *search tree*. First, we discuss the correspondence between a set and a search tree considered abstractly. Then we discuss how the tree can be represented on a computer.

A search tree is a rooted tree, like the execution tree of Figure 10.4. Each node of the tree holds one element of the set being stored. As in Section 10.3, we will consider the tree to be growing downward. Each node can have upto two branches growing downward, denoted *left*, and *right*. It is customary to call the node reached using the left branch the *left child*, and likewise the node reached by the right branch, the *right child*. The children themselves can be considered to be the roots of the (sub) trees beneath them. A node which has no subtree beneath it is called a *leaf*. Here is the key property that we require for a tree storing elements to be a search tree:

$$\begin{array}{c}\text{Values of elements in the}\\ \text{left subtree of node } v\end{array} \quad < \quad \begin{array}{c}\text{Value of element}\\ \text{at node } v\end{array} \quad < \quad \begin{array}{c}\text{Values of elements in the}\\ \text{right subtree of node } v\end{array}$$

An empty set will be represented by an empty tree, i.e. a tree which contains no nodes.

Figure 24.1 shows examples of search trees (parts (a),(b)) and a non-search tree (part (c)). In each case, we have shown the value of the element stored at each node. Thus the

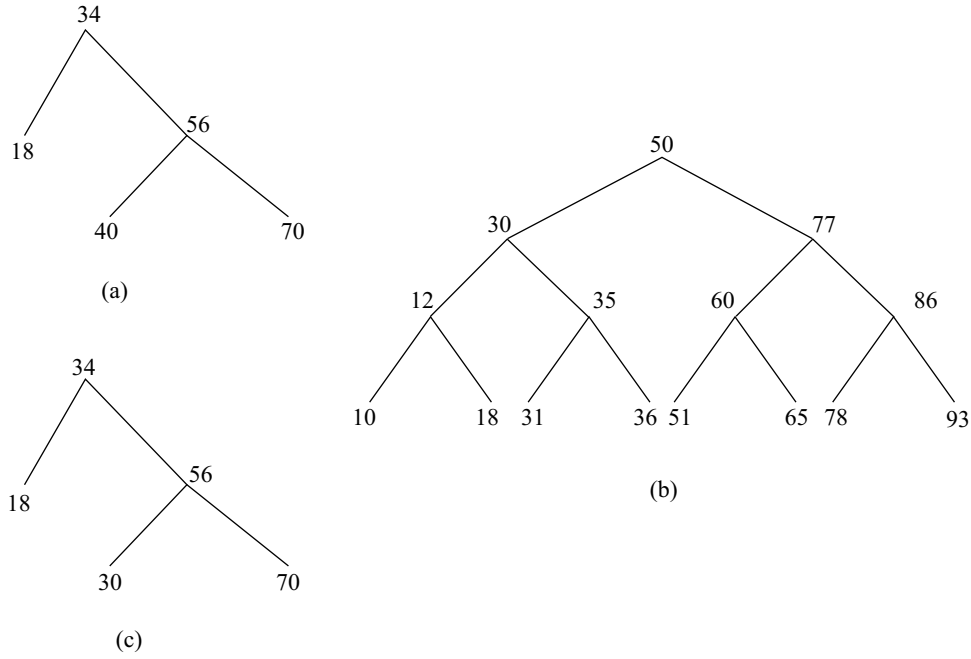


Fig. 24.1 *Examples (a),(b) and non-example (c) of search trees*

tree in (a) would represent the set $\{18, 34, 40, 56, 70\}$, while the one in (b) would represent $\{10, 12, 18, 30, 31, 35, 36, 50, 51, 60, 65, 77, 78, 86, 93\}$. In (c), the subtrees beneath the node with element 34 do not satisfy the search tree property: the right subtree is required to contain elements larger than 34 but it actually contains the element 30. So the tree in (c) will not represent any set, as per our scheme.

Search trees are often called *binary* search trees because each node can have at most two children.

24.1.2 Implementation

A rooted tree is an example of a graph we discussed in Section 23.1 and we can represent rooted trees in the style of Section 23.2. Thus, we will represent nodes by objects of class `Node`, and edges by pointers.

```
struct Node{
  Node *left, *right;
  int value;
};
```

As we said, each node is to store an element of the set, and this will be stored in the member `value` in `Node`. The member `left` will hold a pointer to the left child if a left child exists, else it will be `NULL`. Similarly, for the member `right`. Each node is also connected to its parent i.e. the node (if any) whose child it is. However, we will not have a pointer to the parent, because it will not be needed in the processing we expect to perform.

A search tree (and hence the associated set) is accessed through a pointer to the root node of the tree storing the elements of the set. The pointer will be made `NULL` if we wish to represent an empty tree,

which as we have said represents an empty set. For convenience, we will say the pointer itself represents the tree or the set; though in reality it merely points to objects which are the actual representation.

We will have functions which perform the operations of insertion and membership query. We will give the code for these shortly, but here is how we expect them to be used in a main program.

```
int main(){
  Node* myset=NULL;
  insert(myset,40);
  insert(myset,20);
  cout << "Finding 30: " << find(myset,30) << endl;
  insert(myset,60);
  insert(myset,30);
  cout << "Finding 30: " << find(myset,30) << endl;
}
```

To create a set called `myset`, we simply declare it to be a `Node*` as in the first line. Notice that we have initialized `myset` to `NULL`, indicating that the set is empty.

The next two lines respectively insert the integers 40 and 20 into `myset`. Then the `find` call checks whether the integer 30 is present in `myset`. Clearly, this should return `false`. The next two lines respectively insert 60 and 30 into `myset`. Finally, we check again whether 30 is present. This time `true` should get printed.

Next we discuss the insert function. At the time of the very first call, `myset` points to `NULL`, since it is representing the empty set. We would like `myset` to change as a result of the insertion, and start pointing to a node containing the value being inserted, 40. Clearly, the set (represented by a pointer of type `Node*` to the root) must be passed by reference to `insert`. So here is the code for `insert`.

```
void insert(Node* &pRoot, int elt){
// insert elt into the tree root pointed to by proot.
  if(pRoot == NULL){
    pRoot = new Node;
    pRoot->left = pRoot->right = NULL;
    pRoot->value = elt;
  }
  else{
    if(elt == pRoot->value) return;
    if(elt < pRoot->value) insert(pRoot->left, elt);
    else insert(pRoot->right, elt);
  }
}
```

The algorithm is recursive. The base case is when the tree is empty. In this case, the we must create a node containing the element being inserted; and the root must point to that. If the tree is not empty, then we must modify the existing tree to hold the element being inserted. Note that while doing this, we must maintain the search tree property. Thus, the new element must be inserted in the left subtree if the value at the root is larger than the element being inserted. If the root value is smaller, then the element must be inserted in the right subtree. Finally, if the root value equals the element being inserted, then nothing needs to be done. This is what the above code does.

Note the importance of passing the root to `insert` by reference even in the recursive calls. If in some recursive call, some node `v` has `left` (or `right`) set to `NULL`, and we wish to insert into the corresponding subtree, then after the insertion `v.left` must change. Hence, `v.left` must indeed be passed by reference.

We next consider `find`.

```
bool find(Node* &pRoot, int elt){
// determines whether elt is present in the tree root pointed to
// by proot.
  if(pRoot == NULL) return false;
  if(elt == pRoot->value) return true;
  if(elt < pRoot->value) return find(pRoot->left, elt);
  return find(pRoot->right, elt);
}
```

The function `find` returns `true` if and only if the parameter `elt` is in the tree pointed to by `root`. Its code is again recursive. The base case is if the tree is empty; if so we should clearly return `false`. Otherwise, we compare `elt` to the element at the root. If the two are equal, then we have found `elt` and we return `true`. Otherwise, we must search further. However, note that the search tree property ensures that we need to search only the left subtree if `elt` is smaller than the element at the root, and only the right subtree if `elt` is larger than the element at the root. This is what the code does.

The exercises ask you to define other operations, e.g. printing the set. As you might guess, most operations on trees can be naturally tackled using recursion.

24.1.3 On the Efficiency of Search Trees

We explain why we expect `find` and `insert` as above to run fast on search trees.

As an example, suppose we have in memory the tree in Figure 24.1(b). Suppose we want to find if $x = 63$ is in the tree. Then we would compare x to the number at the root, 50. Finding that x is bigger, we would decide that we only need to search the right subtree. So next we compare x with the number stored in the root of the right subtree, which is 77. This time we realize that x which we are looking for is smaller. So we know we must search the left subtree beneath 77. So we follow the left pointer this time and get to the node containing the key 60. This time we check and realize that x is in fact larger. So we follow the right branch out of the node containing number 60. So we get to the node containing the number 65. Since x is smaller than 65, we know we must go to the left subtree. But there is no left subtree for the node containing 65! So in this case, we have determined that our number x is not present in the set. So we return `false` as the answer.

Notice that we have been able to get to an answer by examining a very few nodes: those nodes containing 50, 77, 60, and 65. We did not examine the other nodes, yet we deduced that the number $x = 63$ could not have been present in the other nodes: because we know that the tree obeys the search tree property.

Of course, the argument given above depends very much upon the shape of the tree in which the set was stored. The tree of Figure 24.1(b) was balanced, i.e. both subtrees under each node had exactly the same number of nodes. If the tree is unbalanced, then the efficiency can become much worse. For example, consider another tree containing the same element as Figure 24.1(b). Suppose that this new tree was built up by first inserting the smallest, then the next larger, and so on. As you will see, the smallest will be at the root, the next larger will be at its right child, the next larger will be at its right

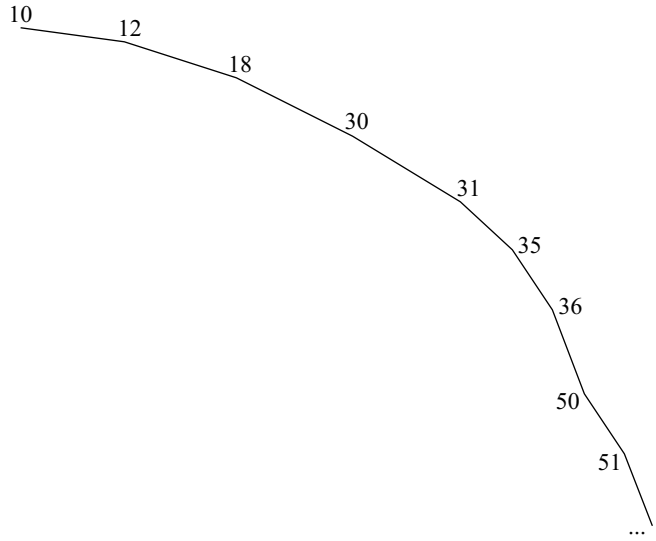


Fig. 24.2 *Another tree containing same elements as in Figure 24.1(b)*

child, and so on. Our "tree", if we could call it that would be a long rightward going path, as shown in Figure 24.2.

In this case, if you want to search for any element, all elements in the set which are smaller will be examined. Thus the time would be much larger.

Note that the execution of `insert` is very similar to that of `find`. As you can see, the first step in insertion is to really go to the place in the tree where the element would have been if it was already present. Thus, the number of tree nodes examined again depends upon the shape of the tree.

More accurately, in any `find` or `insert` operation, we examine all the nodes in some root to leaf path. Obviously, if all such paths are short (i.e. the tree is balanced), then the time will be small. It turns out that this is likely to be the case under some reasonable assumptions. More formally, define the *height* of a tree as the length of the longest root leaf path in a tree. Then the following holds.

Theorem 3 *Suppose numbers from a certain set S are inserted into a search tree using our* `insert` *function. Then if the order to insert is chosen at random, then the expected height of the tree is smaller than $2 \ln |S|$, i.e. twice the natural log of the number of elements in the set.*

The proof of the theorem is outside the scope of this book, but an exercise asks you to validate it experimentally.

Let us try to understand what the theorem says using an example. Suppose we have a set with size 1000, whose elements are inserted in random order into our tree. Then on the average we expect to see that the height will be at most $2 \ln 1000 \leq 14$. Thus, when we perform membership queries (or further insertions) we expect to compare the given number with the numbers in at most 15 nodes in the tree.

You could also ask what are the worst and best heights possible for 1000 nodes. Clearly, if the numbers came in increasing order, then we would get just one path of length 1000 – that would be the height. The other extreme is a tree in which we keep on inserting nodes as close to the root as possible. So we would start by inserting two nodes directly connected to the root, then two nodes connected to each of these, and so on, till be inserted 1000 nodes. So we would have 1 node (the root itself) at

distance 0, 2 nodes at distance 1, 4 nodes at distance 2 and so on till 256 nodes at distance 8, and the remaining $1000 - 256 - 128 - \cdots - 1 = 489$ nodes at level 9. So the height of this tree would be 9.

So it is nice to know that on the average we are likely to be much closer to the best height rather than the worst. Or alternatively, on the average our `find` and `insert` functions will run fast.

24.1.4 Packaging Search Trees/Sets

The code we have developed for representing search trees (and thereby sets) is functionally correct. However, from a stylistic point of view it leaves a lot to be desired. For example, the basic declaration of a set: `Node* myset = NULL;` is somewhat ugly. It would be much nicer if we could instead declare:

```
Set myset;
```

which clearly states that `myset` is a set. We show next how this can be done.

We will define a class `Set` which will have a data member `pRoot` which will hold a pointer to the root of the tree containing the set. The class will have constructor which initializes to the empty set, and member functions `insert` and `find`.

```
struct Node;            // forward declaration

class Set{
  Node *pRoot;
public:
  Set(){pRoot=NULL;}  // initialized to empty set.
  void insert(int elt);
  bool find(int elt);
};

struct Node{
  Set left, right;
  int value;
  Node(int v){ value = v; }
};
```

You will notice that we have changed the type of the members `left, right` in our `struct Node` to `Set` from the earlier `Node*`. Note that a `Set` object is essentially a `Node*` because it only contains one data member of type `Node*`. However by defining `left, right` to be of type `Set`, we can make recursion work in the implementation of member functions `insert` and `find`.

```
bool Set::find(int elt){
  if(pRoot == NULL) return false;
  else{
    if(elt == pRoot->value) return true;
    else if(elt < pRoot->value) return pRoot->left.find(elt);
    else return pRoot->right.find(elt);
  }
}
```

```
void Set::insert(int elt){
  if(pRoot == NULL){pRoot = new Node(elt);}
  else{
    if(elt < pRoot->value) pRoot->left.insert(elt);
    else pRoot->right.insert(elt);
  }
}
```

As you can see, the functions follow the same logic as before, but are simpler to read because there are no references to pointers etc. The main program is also nicer to read.

```
int main(){
  Set myset;
  myset.insert(40);
  myset.insert(20);
  cout << "Finding 30: " << myset.find(30) << endl;
  myset.insert(60);
  myset.insert(30);
  cout << "Finding 30: " << myset.find(30) << endl;
}
```

Note by the way that the user now does not need to know about `struct Node`. This becomes a part of the implementation which is to be hidden from the user.

24.1.5 Balancing a Search Tree

You might be bothered that the above program will work fast "on the average", but might take very long if you are unlucky. What if the numbers in the set got inserted in ascending order, or some such bad order?

In that case, there are advanced algorithms that try to *balance* the tree as it gets built. This is done by modifying an already built tree, and say changing the root. With such rebalancing, it is indeed possible to ensure that the height of the tree remains small. Further, rebalancing algorithms have been developed that also run very fast. But this is outside the scope of this book.

24.1.6 Search Trees and `map`s

The `(index,value)` pairs constituting a `map` from the C++ Standard Library are stored using binary search trees. The ordering rule is that all pairs in the left subtree must have index smaller than that at the root, which in turn must be smaller than the indices of the elements in the right subtree. Further, the tree is kept balanced as discussed above. Thus making an access such as `map[index]` happens fairly fast, i.e. if n is the number of pairs stored in the map, then the longest path length is proportional to $\ln n$, or equivalently to $\log_2 n$.

24.2 LAYOUT OF MATHEMATICAL FORMULAE

We now consider a very tiny version of the formula-layout problem. Specifically, we will consider formulae in which only the 2 arithmetic operators + and / are used. Our program must take any such

formula, written in a language like the TEX language mentioned at the beginning of the chapter, and produce a layout for it. This layout must be shown on our graphics canvas. As you will discover in the exercises, once you master sum and division, implementing other operators and more complex operations such as summations using the $\sum$ symbol is not much more difficult. Of course, all this will still be far from what TEX accomplishes.[1]

24.2.1 Input Format

The first question, of course, is how we should specify the formula to the program. One possibility is to just use the TEX language. However, that seems too elaborate, after all we only have two operators. Another possibility is to specify the formula in the style used in C++ to specify mathematical formulae, e.g. to get $\frac{a}{b}$ we could supply `a/b` as input.

We will indeed use the C++ style, but with a slight variation. We will require that the formula be specified in the style used in C++, with the operands to the + operator as well as the / operator placed in parentheses.

So as a simple example, whereas in C++ you could write `a/b`, to specify this to our program you would have to write `(a/b)`, because the rule says that the operands to every operator must be inside parentheses. Figure 24.3 gives some more examples. As you can see, the input required by our program is more verbose as compared to what is required to specify the formula in C++. In the exercises, we will explore the issues in allowing less verbose input.

We should note an interesting feature of our input format. You will note that any formula written in the style described above will have one of the following three forms:

- `p` : where `p` is a primitive formula, i.e. an identifier or a number.
- `(L+R)` : where `L` and `R` are themselves formulae. For example, in `((((x+1)/(x+3))+(x/5))+6)` we have `L = ((x+1)/(x+3))+(x/5))` and `R = 6`
- `(L/R)` : where `L` and `R` are themselves formulae. For example, in `(a/(b+c))` we have `L = a` and `R = (b+c)`.

	Desired output	Input required by our program
0.	a	`a`
1.	$\frac{a}{b+c}$	`(a/(b+c))`
2.	$a + \frac{b}{c}$	`(a+(b/c))`
3.	$a + b + c + d$	`(((a+b)+c)+d)`
4.	$\frac{x+1}{x+3} + \frac{x}{5} + 6$	`((((x+1)/(x+3))+(x/5))+6)`

Fig. 24.3 *Examples of output and input*

[1] TEX is a complete document processor. Furthermore, even for the purpose of laying out mathematical formulae, it is very sophisticated. For example, it adjusts sizes of the text, which our program will not.

In other words, formulae are built up either using primitive formulae ("base case") or other formulae. Hence, a formula is said to have a recursive structure. The recursive structure will be very useful.

In the input, we must also specify where we want the layout to appear. Should it appear in the center of the graphics canvas, or in the top left corner, or somewhere else? We can think of the layout as contained in a *bounding box*, by which we mean the smallest (axis parallel) rectangle that contains it. For simplicity, we will say that the program should produce a layout such that the top left corner of the bounding box of the layout appears at a given point (x, y), i.e. x, y will be two additional inputs to the program. In what follows, we will use the phrase "draw the formula at (x, y)" to mean "draw the formula such that the top left corner of the bounding box of its layout is at (x, y)".

24.2.2 Representing Mathematical Formulae

The user types in the formula in the format described above. It must then be read by our program and converted into a convenient representation for subsequent processing. For subsequent processing, it will be convenient if we can access the subformulae very easily. For this it is customary to view a formula as a rooted tree, sort of like the execution tree of Figure 10.4, or a search tree discussed earlier. Figure 24.4 shows two formulae drawn as rooted trees. Leaf nodes, i.e. nodes that have no children correspond to primitive formulae. Internal nodes (i.e. nodes that are not leaves) are associated with an operator. A subtree, i.e. any node and all the nodes below it, represents a subformula used to build up the original formula. For example, in Figure 24.4(a), the subtree including and beneath the node labelled + corresponds to the subformula $b + c$. Similarly, in Figure 24.4(b), the node labelled / on the left side and the nodes below it correspond to the subformula $\frac{x+1}{x+3}$, whereas the node labelled / on the right and the nodes below it together correspond to the subformula $\frac{x}{5}$.

Thus, our program will read in a formula, and represent it in memory as a rooted tree. The rooted tree will be represented as per the discussion in Section 24.1.2, i.e. the nodes will be represented by structures and the edges by pointers. The structure representing a node will hold the information associated with a node, e.g. whether the node is associated with an operator, and if so which one, or whether the node is associated with a primitive formula, and if so which one. And of course the node will contain pointers to nodes that are connected to it. So we define a structure `Node` as follows.

```
struct Node{     // we give member functions later.
  char op;       // operator associated with the node if any.
  string value; // primitive formula associated with the node if any.
```

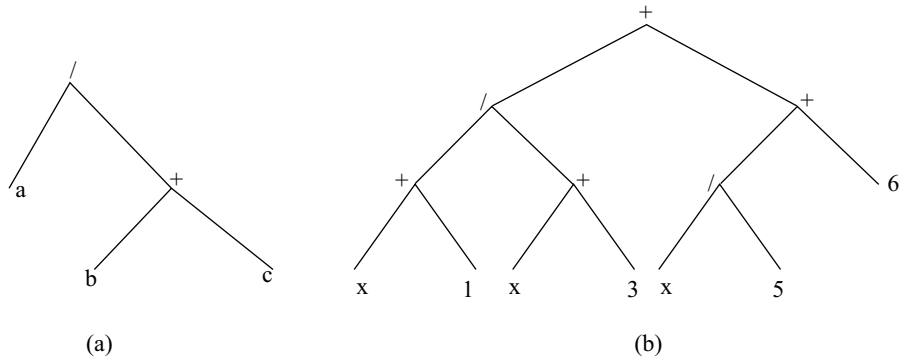


Fig. 24.4 *(a) Tree for $\frac{a}{b+c}$ (b) Tree for $\frac{x+1}{x+3} + \frac{x}{5} + 6$*

```
  Node* L;     // pointer to left subformula if any.
  Node* R;     // pointer to right subformula if any.
};
```

This structure can be used to represent primitive as well as non-primitive formulae. If a formula is primitive, i.e. consists of an identifier or a number, we will store that symbol or number in the member `value` as a character string. Otherwise, the formula must be a binary composition of two smaller formulae. In this case, we store the operator in the `op` member, and we store pointers to the roots of the subformulae in the members `L,R`. A node may also have a connection to a parent node; however we have chosen not to have a pointer to the parent node because it will not be needed in the processing that we will do on nodes.

It will be convenient to have two constructors for nodes. The first constructor is useful for constructing nodes corresponding to primitive formulae.

```
Node::Node(string v){        // primitive formula constructor
  value = v;
  op = 'P';    // convention: 'P' in op denotes primitive formula.
  L = NULL;
  R = NULL;
}
```

A primitive formula does not have subformulae, and hence we have set `L, R` to `NULL`. Here is how we can call this constructor.

```
Node aexp("a");
Node bexp("b");
Node cexp("c");
```

This creates nodes `aexp, bexp, cexp` corresponding to primitive formulae a, b, c. To combine these formulae into bigger formulae we need another constructor.

```
Node::Node(char op1, Node* L1, Node* R1){
        // recursive constructor
  value = "";
  op    = op1;
  L   = L1;
  R   = R1;
}
```

The arguments to the constructor are pointers `L1, R1` to nodes representing subformulae, and the operator `op1` using which the subformulae are expected to be combined. As an example, to construct a representation for $b + c$, we will need the operator to be $+$, and `L, R` to be (pointers to) nodes representing b, c. But we have already created `bexp, cexp` for this purpose. Thus, we can construct the tree representing the expression $b + c$ by writing

```
Node bplusc('+', &bexp, &cexp);
```

Finally to represent $\frac{a}{b+c}$ it suffices to write

```
Node f1('/', &aexp, &bplusc);
```

Thus, `f1` will be the root of the tree for the formula $\frac{a}{b+c}$. Thus, we can say that `f1` represents the formula. Or alternatively, we can also construct a representation more directly:

```
Node f2('/', new Node("a"),
             new Node('+', new Node("b"), new Node("c"))
         );
```

An important point to note here is that the operator `new` when used on a constructor call returns a pointer to the constructed object, which is exactly what we want as an argument to our recursive constructor. Thus, `f2` will also be a root of the tree for the formula $\frac{a}{b+c}$, and can thus be said to represent the formula.

There is a difference between the two constructions, however. In the first construction, all memory for the formula comes from the current activation frame. In the second construction, all memory except for the node `f2` comes from the heap, the memory for `f2` comes from the current activation frame.

Once we have a representation for formulae, our task splits into two parts:

1. Read the formula from the input in the format specified in Section 24.3 and build a representation for it using the `Node` class.
2. Generate the layout from the constructed representation. It is natural to define member functions on `Node` which will generate the layout.

We consider these steps in turn.

24.2.3 Reading in a Formula

For simplicity, we will make two assumptions: (a) Each number or identifier is exactly one character long, and (b) there are no spaces inside the formula. In the exercises you are asked to write code which allows longer primitive formulae and also spaces.

We read the formula character by character and build a representation as we go along. To read a character from a stream `infile`, we can use the operation `infile.get()` which returns the next character as an integer value.

If the very first character read is a number or a letter, then we have indeed read a primitive formula and we can stop.

If what is read is the character `'('`, then we know that the user is supplying us a non-primitive formula. In that case we know that we must next see in succession (a) the left-hand-side formula, (b) an operator, and (c) the right-hand-side formula. To read in the left-hand-side formula, we merely recurse! After the formula has been read, we read a single character, and it had better be an operator, as per (b). After that we recurse again, in order to get the right-hand-side formula, as per (c)! And after that we must see a ')' to match the initial open parenthesis. We have written this as another constructor. Note that our code is extremely simple but it can read in very complicated formulae.

```
Node::Node(istream &infile){
  char c=infile.get();
  if((c >= '0' && c <= '9') ||      // is it a primitive formula?
     (c >= 'a' && c <= 'z') ||
     (c >= 'A' && c <= 'Z')){
    L=R=NULL; op='P'; value = c;
  }
```

```
  else if(c == '('){       // does it start a non-primitive formula?
    L = new Node(infile);// recursively get the lhs formula
    op = infile.get();     // get the operator
    R = new Node(infile);// recursively get the rhs formula
    if(infile.get() != ')')
      cout << "No matching parenthesis.\n";
  }
  else cout << "Error in input.\n";
}
```

Now we can write a part of the main program.

```
int main(){
  Node f(cin);
}
```

This will call our latest constructor with the parameter `infile` being `cin`, i.e. the formula will be read from the keyboard. You may type in something like

```
(a/(b+c))
```

and it will create `f`, just as we created `f2` earlier.

24.2.4 The Drawing Algorithm

The input to the drawing step will be a formula F (specified by the root of its tree representation), and numbers x_F, y_F. The requirement is that the formula F be drawn at (x_F, y_F), i.e. so that the top left corner of the bounding box of the layout of F is at (x_F, y_F).

It is natural to consider recursion. If F is a primitive formula, we should draw it directly at a suitable place. If F has the form L/R or $L + R$, then we will recursively draw f, g, and also draw the horizontal bar or the $+$ symbol as needed. Figure 24.5 shows this general scheme. We have not used C++ code in this figure so as to make it easier to write subscripts etc. In order to make this general scheme work, we need to determine values of the various coordinates, i.e. x_p, y_p and so on.

Draw(F, x_F, y_F){ // Draws F at (x_F, y_F)

1. If F is some primitive formula p, then write the text of F at a suitably determined point (x_p, y_p).
2. Else if F has the form L/R, then determine x_L, y_L, x_R, y_R suitably and call Draw(L, x_L, y_L), Draw(R, x_R, y_R). Also determine x', y', x'', y'' and draw a line from $(x,' y')$ to (x'', y'').
3. Else if F has the form $L + R$, then determine x_L, y_L, x_R, y_R suitably and call Draw(L, x_L, y_L), Draw(R, x_R, y_R). Also determine x', y' and draw the symbol $+$ at (x', y').

}

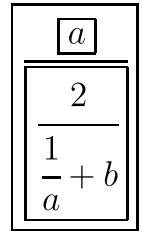
Fig. 24.5 *General recursive strategy for drawing*

You should take a minute to reflect on the strategy given in Figure 24.5. True, we have not yet specified how to determine all the numbers $x_p, y_p, x_L, y_L, x_R, y_R \ldots$. Nevertheless, it is worth admiring the elegance of this strategy: it only has three relatively simple-looking steps, and will yet be able to draw arbitrarily large and complex formulae! This is the power of recursion.

Next we consider how to determine the numbers $x_p, y_p, x_L, y_L, x_R, y_R \ldots$ of Figure 24.5.

If the formula being drawn is primitive, i.e. $F = p$ where p is a primitive formula, we can determine its width w_p and height h_p using the functions `textWidth` and `textHeight` from Section 5.3.4. Then we just ask the text for p to be written centered at $(x_F + w_p/2, y_F + h_p/2)$. This is the base case of the recursion (case 1 of Figure 24.5) and it is easily handled.

Next we consider the second case of Figure 24.5, i.e. $F = L/R$. It is worth taking an example. Say $L = a$ and $R = \frac{2}{\frac{1}{a}+b}$. Then the layout is



We have drawn the bounding boxes of L, R and also of L/R. The boxes of L, R will really be in contact with the box of L/R, we have drawn them separated for ease of understanding. As you can see, the formula L must be at the top. Given that (x_L, y_L) denote the coordinates where L is to be drawn, we have

$$y_L = y_F \tag{24.1}$$

Below L, there must be a horizontal bar denoting the division, below which there must be the formula R. Thus, given that given that R is to be drawn at (x_R, y_R), we have

$$y_R = y_F + h_L + h_- \tag{24.2}$$

In this, h_L denotes the height of the layout of L when drawn out completely, and h_- denotes the vertical space needed to accommodate the horizontal bar. See Figure 24.6(b). Thus, we must determine the height of the layout of L before we can draw F. We will see how to do this shortly. To find x_L, x_R, we first note that L, R must be aligned so that their centers are on the same vertical line. Further, we can see (Figure 24.6(b))

$$w_F = \max(w_L, w_R) \tag{24.3}$$

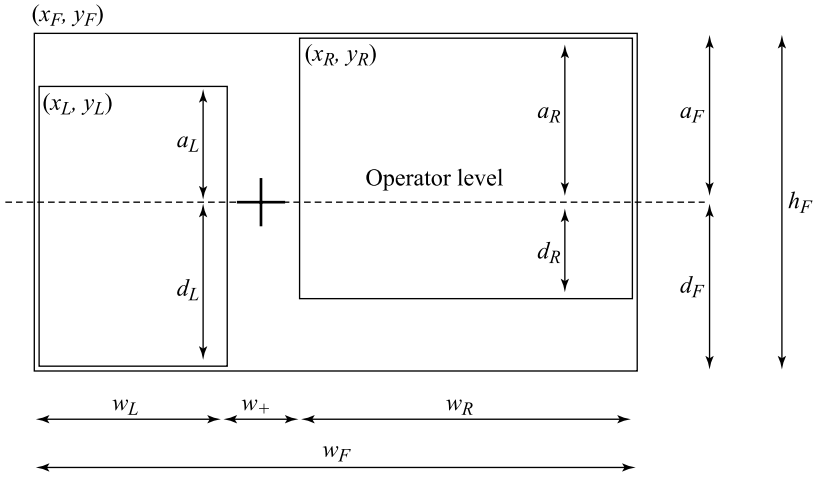
where w_F, w_L, w_R are the widths of the layouts of F, L, R. Thus, the x coordinate of the centers of F, L, R must be at $x_F + w_F/2$. Thus,

$$x_L = x_F + w_F/2 - w_L/2 \tag{24.4}$$

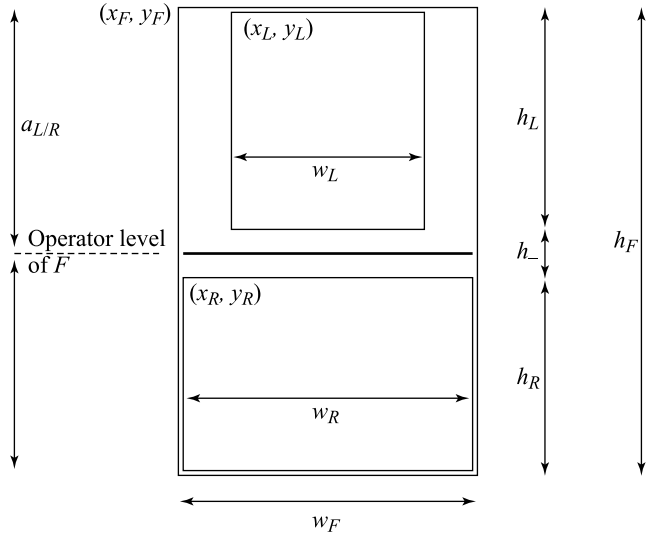
$$x_R = x_F + w_F/2 - w_R/2 \tag{24.5}$$

As for drawing the horizontal bar, it is easy to see that

$$x' = x_F \tag{24.6}$$



(a) Layout of $F = L + R$



(b) Layout of $= L/R$

Fig. 24.6 *Composing layouts*

$$y' = y_F + h_L + h_-/2 \tag{24.7}$$

$$x'' = x_F + w_F \tag{24.8}$$

$$y'' = y' \tag{24.9}$$

Thus, if only we know the widths and heights of L, R we will be able to implement case 2 of Figure 24.5.

Next consider case 3 of Figure 24.5, i.e. $F = L + R$. As an example, as before, let $L = a$ and $R = \frac{2}{\frac{1}{a}+b}$. Then the layout of $F = L + R$ is:

$$\boxed{\boxed{a} + \boxed{\frac{2}{\frac{1}{a} + b}}}$$

As before, we have shown the bounding boxes for L, R as well as F. Now we can see that the layout of L is leftmost inside the layout of F. Thus,

$$x_L = x_F \tag{24.10}$$

The layout of R begins at a distance $w_L + w_+$, where the w_+ is the width needed to layout the $+$ symbol. See Figure 24.6(a). Thus,

$$x_R = x_F + w_L + w_+ \tag{24.11}$$

Determining y_L, y_R is a little tricky. For this, we need to understand how L, R and the symbol $+$ align with each other. In our example, the center of L which is just the primitive symbol a aligns with the horizontal bar of $+$. On the other hand, the horizontal bar of $+$ does not align with the center of R, but instead aligns with the horizontal bar separating the numerator and the denominator of R. So it seems that with each layout is associated a horizontal line, which we will call the *operator level*, which must align with the horizontal bar of the $+$. Clearly, the operator level for a primitive formula passes through its center, whereas that for a ratio passes through the horizontal bar inside it.

So this points to a complication. We need to know the position of the operator level for each formula, and not just the height and width. For a formula f, let its *ascent* denoted as a_f be the distance to which the formula rises above its operator level. Let its *descent* denoted as d_f be the distance to which the formula dips below its operator level. As can be seen from Figure 24.6(a), when we align the operator levels, the formula $F = L + R$ will rise above the operator line by a distance equal to the maximum of the ascents of the L, R. In other words, we have

$$a_F = \max(a_L, a_R) \tag{24.12}$$

Similarly,

$$d_F = \max(d_L, d_R) \tag{24.13}$$

Noting that the operator level of L, R align, we have

$$y_L = y_F + a_F - a_L \tag{24.14}$$

$$y_R = y_F + a_F - a_R \tag{24.15}$$

Finally, we need to decide where the $+$ needs to be placed. From Figure 24.6(a), we see that the coordinates are:

$$x' = x_F + w_L + w_+/2, \tag{24.16}$$

$$y' = y_F + a_F \tag{24.17}$$

Thus, we are now in a position to implement step 3 of Figure 24.5 if only we could determine the width, height, ascent and descent of the various formulae. Note by the way that the ascent, descent and height are related. For any formula f, $h_f = a_f + d_f$.

So the question is, can we determine the height, width, ascent, and descent of each subformula in our target formula? We do know some additional relationships between these quantities. For example, we know that

$$w_{L+R} = w_L + w_+ + w_R \tag{24.18}$$

This is immediate from Figure 24.6(a). Notice that equations (24.3) and (24.18) express the width of a formula in terms of the widths of its subformulae. Thus, we can compute this by recursion! Likewise equation (24.12) expresses the ascent of $L + R$ in terms of a_L, a_R. We could write this recursively if we could determine $a_{L/R}$ as well. From Figure 24.6(b), this is seen to be

$$a_{L/R} = h_L + h_-/2 \tag{24.19}$$

Likewise,

$$d_{L/R} = h_R + h_-/2 \tag{24.20}$$

Thus, we have the ascents and descents also expressed in terms of the ascents and descents of the subformulae. So we can determine these by recursion also.

So now we are ready to write the code. Our drawing code will be in the spirit of Figure 24.5. However, but before calling draw, we will call another recursive function which will determine widths, heights, ascents and descents.

24.2.5 Implementation

We will write a member function called `setSizes` in `Node` that will determine height, width etc. For drawing we will have a member function `draw`, along the lines of Figure 24.5. Here is the new declaration of `Node`.

```
struct Node{
  static const int h_bar = 10; // space for horizontal bar
  Node *L, *R;
  char op;
  string value;
  double width, height, ascent, descent;
  Node(string v);
  Node(char op1, Node* L1, Node* R1);
  Node(istream& infile);
  void setSizes();
  void draw(double clx, double y); // to actually draw
};
```

Note that `h_bar` is h_- in the discussion above. Further, we have added data members to hold the width, height, ascent, descent for each formula denoted by the subtree under the node.

We have already given the implementations for the constructors. The member function `setSizes` will calculate the values of width, height, ascent, descent using the recursive ideas described above. Here is the code for it.

```
void Node::setSizes(){
  switch (op){
```

```
  case 'P':                 // Primitive formula
    width = textWidth(value);
    height = textHeight(); ascent = descent = height/2;
    break;
  case '+':                  // case L+R
    L->setSizes();
    R->setSizes();
    width = L->width + textWidth(" + ") + R->width;
    descent = max(L->descent, R->descent);
    ascent = max(L->ascent, R->ascent);
    height = ascent + descent;
    break;
  case '/':                  // case L/R
    L->setSizes();
    R->setSizes();
    width = max(L->width, R->width);
    ascent = h_bar/2 + L->height;
    descent = h_bar/2 + R->height;
    height = ascent + descent;
    break;
  default: cout << "Invalid input.\n";
  }
}
```

This follows the ideas of Section 24.2.4. The first case, when the formula is primitive, is straightforward. For the case when the formula is a sum, we first call `setSizes` on `R` and `L`, i.e. calculate height, etc. After this, we use Eq. (24.18) to calculate the width, Eq. (24.12) to calculate the ascent, and Eq. (24.13) to calculate the descent. The height is set to the sum of the ascent and the descent. For the final case, i.e. the formula is a ratio, we again first calculate the height etc. for the subformulae. After this we use equation (24.3) to compute the width, equation (24.19) to calculate the ascent, and equation (24.20) to calculate the descent. The height is set as the sum of the ascent and the descent.

The member function `draw` actually does the drawing, using the outline from Figure 24.5.

```
void Node::draw(double xF, double yF){
  switch(op){
  case 'P':
    Text(xF + width/2, yF + ascent, value).imprint();
    break;
  case '+':
    L->draw(xF, yF + ascent - L->ascent);
    R->draw(xF + L->width + textWidth(" + "), yF + ascent
              - R->ascent);
    Text(xF + L->width + textWidth(" + ")/2, yF + ascent,
         string(" + ")).imprint();    // draw the '+' symbol
    break;
  case '/':
    L->draw(xF + width/2 - L->width/2, yF);
```

```
    R->draw(xF + width/2 - R->width/2, yF + L->height + h_bar);
    Line(xF, yF + ascent, xF + width, yF + ascent).imprint();
                                        // horizontal bar
    break;
  default: cout << "Invalid input.\n";
  }
}
```

The case when the formula is primitive is straightforward. For the case `'+'` above, L is drawn at the coordinates as given by equations (24.10) and (24.14). Then R is drawn at coordinates given by Eq. (24.11) and (24.15). The position of the $+$ is as per equations (24.16) and (24.17). The case when the formula is a ratio is also as per the discussion of Section 24.2.4. L is drawn at coordinates as per equations (24.4) and (24.1). Likewise, `R` at coordinates as per equations (24.5) and (24.2). Finally, the horizontal bar is drawn as per equations (24.6–24.9).

24.2.6 The Complete `main` Program

Here is the main program.

```
int main(){
  initCanvas("Formula drawing");

  Node e('+', new Node("1"),
         new Node('/', new Node("2"),
             new Node('+', new Node('/', new Node("451"),
                  new Node("5")), new Node("35"))));

  e.setSizes();
  e.draw(50,50);

  Node g(cin);
  g.setSizes();
  g.draw(250,50);

  getClick();
}
```

The formula `e` is specified as a part of the program and then drawn. The formula `g` on the other hand is read from the keyboard. Suppose you typed

```
((((x+1)/(x+3))+(x/5))+6)
```

to the program. Then you would get the drawing shown in Figure 24.7.

$$1+\cfrac{2}{\cfrac{451}{5}+35} \qquad \frac{x+1}{x+3}+\frac{x}{5}+6$$

Fig. 24.7 *Layout of some formulae*

We should perhaps use the name `Formula` instead of `Node`, to indicate its function rather than implementation. Since we are unlikely to need to represent an empty formula, it is not necessary to package formulae in the style we packaged sets in Section 24.1.4.

24.2.7 Remarks

Recursive structures appear in many real-life situations. For example, the administrative heirarchy of an organization is recursive, e.g. there is a director/president/prime minister, to whom report deputies, to whom report further deputies.

It is natural to associate a tree with a recursive structure. The substructures are denoted as subtrees, and the element joining the subtrees, e.g. the director, will correspond to the root. In the case of mathematical expressions, the operator corresponds to the root, and the sub expressions correspond to the subtrees.

You may be wondering why we require that the formulae to be layed out be specified in our verbose format; why not just specify them as they might be in C++? It turns out that getting a program to read formulae in C++ like languages is a classical computer-science problem, in its most general setting. If you pursue further education in computer science, you will perhaps study it in a course on compiler construction, or automata theory. For now suffice it to say that reading C++ style expressions is a difficult problem. However, in the exercises you are encouraged to think about it.

EXERCISES

1. Add a `print` member function to the `Set` class so that elements of a set can be printed. *Hint*: use recursion: first print the members in the left subtree, then the value stored at the current node, and then the value in the right subtree.
2. Your answer to the previous problem will likely print absolutely nothing for an empty set. Suppose that you are to print a message "Empty set" in such cases. *Hint*: Use one non-recursive member function which calls a recursive one.
3. Add a member function with signature `int smaller(int elt)` which returns the number of elements in the set smaller than `elt`. *Hint*: Add a member `count` to each node which will indicate the number of nodes in the subtree below that node. You will need to update `count` values suitably whenever you insert elements. Now use the `count` value to respond to `smaller`.
4. Experimentally verify Theorem 3. Let n denote the number of elements in the set. Assume without loss of generality that the elements in the set are integers $1, 2, \ldots, n$. Run the insertion algorithm by generating numbers between 1 and n (without replacement) in random order. Measure the height of the resulting tree. Repeat 100 times and take the average. Repeat for different values of n and plot average tree height versus n.
5. Modify the code of the `Set` class so that the allocated memory is returned to the heap whenever a `Set` object is destroyed. Do this by writing an appropriate destructor. For this, you will need to note that when a destructor is called on an object, it will cause destructors to be called on the members of that object as well. Also note that if you delete an object its destructor will automatically be called.

6. Extend the formula-drawing program so that it allows the operators `'*'`, `'+'` and `'-'`. This is not entirely trivial: make sure your program works correctly for input `((x+3)*(x-2))`. You will see that you may need to add parenthesization to the output. For simplicity, you could parenthesize every expression when in doubt.

7. Add an operator `'^'` to denote exponentiation in the formula drawing program. In other words, allow `Node('^',Node("x"), Node("y"))` which should be drawn as x^y.

8. In Section 24.2, we assumed that the input will be given with no spaces in it. Modify the code developed in that section so that the user may include spaces in the input, say between operators and operands.

9. The expression `infile.peek();` returns the next character in the file without actually reading it. Use this to modify the code of Section 24.2 further so as to allow primitive expressions to be longer than a single character.

10. How will you represent integration with lower and upper limits, and the integrand? In other words, you should be able to draw formulae such as

$$\int_0^1 \frac{x^2 dx}{x^2+1}$$

Hint: The best way to do this is to use a ternary operator, say denoted by the letter `I`, which takes as arguments 3 formulae: the lower limit `L` , the upper limit `U`, and the expression `E` to be integrated. You could require this to be specified as `(L I U E)`.

11. As we have defined, our formulae cannot include brackets. Extend our program to allow this. You could think of brackets being a unary operator, say `B`. Since it is our convention to put the operator second, you could ask that if a formula `F` is to be bracketed, it be written as `(F B)`. Make sure that you draw the brackets of the right size.

12. You may want to think about how the program might change if the formula to be layed out is specified in the standard C++ style, i.e to draw $a + \frac{b}{c}$ the input is given as `a+b/c` rather than `(a+(b/c))` as we have been requiring. The key problem as you might realize, is that after reading the initial part `a+b` of the input, you are not sure whether the operator + operates on `a,b`. This is the case if the subsequent operator, if any, has the same precedence as +. However, if the subsequent operator has higher precedence, as in the present case, then the result of the division must be added to `a`. So you need to *look ahead* a bit to decide structure to construct. This is a somewhat hard problem, but you are encouraged to think about it. Note that your job is not only to write the program, but also argue that it will correctly deal with all valid expressions that might be given to it.

13. Add a `deriv` member function, which should return the derivative of a formula with respect to the variable `x`. Use the standard rules of differentiation for this, i.e.

$$\frac{d(uv)}{dx} = v\frac{du}{dx} + u\frac{dv}{dx}$$

This will of course be recursive. You should be able to draw the derivatives on the canvas, of course.

14. You will notice that the result returned by `deriv` often has sub-expressions that are products in which one operand is 1 and sums in which one operand is 0. Such expressions can be simplified. Add a `simplify` member function which does this. This will also be recursive.

CHAPTER 25

Inheritance

Inheritance is one of the most important notions in object-oriented programming. The key idea is: you can create a class `B` by designating it to be *derived* from another class `A`. Created this way, the class `B` gets or *inherits* all the data members and ordinary function members of `A`. In addition to what is inherited, it is possible to include additional data and function members in `B`. It is also possible to redefine some of the inherited function members. The class `B` thus created, is said to be a *subclass* of the class `A`. As you might suspect, this is a convenient way to create new classes.

The most common and natural use of inheritance is in the following setting. Suppose, a program deals with categories of objects, which are divided into subcategories. For example, a program might be concerned with bank accounts, and these may be divided into different types of accounts, e.g. *savings* accounts and *current* accounts. Or a program might be concerned with drawing geometric shapes on the screen, and the category of shapes, as we have seen, might be subdivided into subcategories such as circles, lines, polygons and so on. In such cases it turns out to be useful to represent a category (e.g. accounts or geometric shapes) by a class, and subcategories (savings accounts and current accounts, or lines and circles) by subclasses. As you will note, the attributes associated with a category (e.g. account balance, or screen position) are present in the subcategories. Hence, it is natural that these attributes will be defined in the class corresponding to the category. These attributes will be inherited when we define subclasses corresponding to the subcategories. In each subclass we need additionally define the attributes which are specific to the corresponding subcategory. For example, in the circle subclass we could define the attributes center and radius, while the polygon class will have vertices as the attributes. Categories and subcategories are common in real life, and hence inheritance can play a central role in the design of complex programs.

First some terminology. Suppose we derive a class `B` from a class `A` using inheritance. It is customary to say that class `B` is a *subclass* of class `A`, is *derived* from `A`, or obtained by *extending* class `A`. And of course, `B` is said to *inherit* from `A`. It is likewise customary to say that `A` is a *superclass* of `B`, or *base* class of `B` or sometimes the *parent* class of `B`. We can have several classes say `B,C,D` inheriting from `A`. In turn, we may have classes `E,F` inheriting from `B`. In such a case, the classes `A, B, C, D, E, F` are said to constitute an *inheritance heirarchy*.

In this chapter, we will mainly consider the mechanics of inheritance. We begin by considering a simple example and then discuss how to use inheritance in general. An important aspect of inheritance is that we can have many views of an object; sometime we might consider it to be an instance of a

subcategory (e.g. a circle) at other times we may consider it as belonging to a category (e.g. a shape). In order to be able to shift views smoothly, we need the notions of polymorphism and virtual functions. We discuss these notions.

In the last section, we will discuss an example which will help you see the utility of the machinery we develop. More will come in the next chapter, where we consider how to design programs using inheritance.

25.1 TURTLES WITH AN ODOMETER

Suppose we wish to design a class `mTurtle` (short for metered turtle) which is exactly like the class `Turtle`, except that the turtle will keep a count of how much distance it has covered. So a `mTurtle` will be able to move forward, turn, change colours etc. just like a `Turtle`, but in addition it will have a member function `distanceCovered` which will return the total distance covered till then.

Here is an example of a main program that we would like to write.

```
int main(){
  initCanvas();
  mTurtle m;
  m.forward(100);
  m.right(90);
  m.forward(50);
  cout << m.distanceCovered() << endl;
}
```

This program should print 150.

25.1.1 Implementation using Composition

We first consider how `mTurtle` could be implemented without inheritance, using what you already know. A simple idea is to *compose* an `mTurtle` object by having a `Turtle` object as a member. We will call this class `mTurtleC`.

```
class mTurtleC{   // Solution using composition. (no inheritance).
  Turtle t;
  double distance;
public:
  mTurtleC(){
    distance = 0;
  }
  forward(double d){
    distance += abs(d);  // because d may be negative.
    t.forward(d);
  }
  double distanceCovered(){
    return distance;
  }
  void right(double angle){
    t.right(angle);
  }
```

```
  void left(double angle){
    t.left(angle);
  }
  // similar forwarding code for other functions allowed on Turtle..
};
```

As you can see, inside `mTurtleC`, we have a `Turtle` object which you will see on the screen, and a member `distance` which keeps track of how much the turtle has moved. Clearly, when an `mTurtleC` is created, we should set `distance` to 0, which is what the constructor above does. The constructor does not appear to do much with the turtle member `t`. But you know that the member `t` is also created, using the default `Turtle` constructor. Thus, a turtle will appear on the screen. The member function `forward` in `mTurtleC` causes `distance` to be updated, and causes the turtle to move as well. Finally, the member function `distanceCovered` prints `distance` as expected.

The code for `right` is simple, we just call the function `right` on member `t`, with the same argument. We will have to write such *forwarding* functions for other functions such as `left`, `hide` and so on.

Clearly, this will enable us to write the main program given earlier; we merely have to use `mTurtleC` in it instead of `mTurtle`. The solution is fairly satisfactory, the main drawback is the need to write the forwarding functions.

25.1.2 Implementation using Inheritance

Using inheritance, we can define a metered turtle more compactly, as follows. We will call this class `mTurtleI`.

```
class mTurtleI : public Turtle{  // solution with inheritance
  float distance;
public:
  mTurtleI(){
    distance = 0;
  }
  void forward(float d){
    distance += abs(d);
    Turtle::forward(d);
  }
  float distanceCovered(){
    return distance;
  }
};
```

We will shortly explain what each line in this does. For now, we merely note that this will essentially do what the code in Section 25.1.1 did. With this, we will be able to run the main program given at the beginning of Section 25.1, of course we will need to use `mTurtleI` in the main program instead of `mTurtle`.

Note that we have not defined functions such as `right`. We have not explicitly defined the member `t` of type `Turtle`. As we will see next, these are *inherited!*

25.2 GENERAL PRINCIPLES

In general, we can define a class `B` as a subclass of an existing class `A`, by writing

```
class B : type-of-inheritance A {
  // body describes how B is different from A
}
```

In this, `type-of-inheritance` can either be `public`, `private` or `protected`. We begin our discussion with `public` inheritance, which was used in our class `mTurtleI` in the previous section. We will discuss other types of inheritance in Section 25.7.

Note that the name `A` must itself already have been defined when we define `B`. This is typically accomplished by including the header file of `A`.

The above definition will create a class `B` which starts off with all data members and function members of `A`, except for the constructors and the destructor. If we want `B` to have some additional data or function members, we define them in the body. We can also redefine some functions that were present in `A`—the new definitions will be used for objects of class `B`. We discuss the details next.

As we have said, the data members present in `A` will also appear in the objects of class `B`. They will appear individually, i.e. we will be able to access them directly as per some rules that we discuss soon. However, we can also think that the inherited members together constitute an *inherited object* of class `A` inside each object of class `B`. In addition, we can have new data members, by defining them in the body of the definition of class `B`.

Thus, for our class `mTurtleI` of Section 25.1.2, we will get the inherited object and the new data member `distance`. As you may guess, a `Turtle` object will contain many data members, for example a member `color` (of type `Color`) which holds information about the colour of the turtle. These would be included in of `mTurtleI`. Figure 25.1(a) shows the contents of the object produced for the class `mTurtleI`. For comparison, in Figure 25.1(b), we have also shown the contents of an object of type `mTurtleC` as well. An object of class `mTurtleC` has two data members, a member `distance` which is a `double` and a member `t` which is a `Turtle`. Thus, both objects of Figure 25.1 really contain the same information. The main difference is that in `mTurtleC` the members of the contained turtle object `t` such as `color` are not directly accessible, whereas in `mTurtleI`, these members are directly accessible.

All the member functions in class `A`, except for the constructors and destructor are assumed present in `B`. These functions will refer to members of `A`, but this will not cause a problem because these members are also inherited. The body of the definition can also contain additional member functions

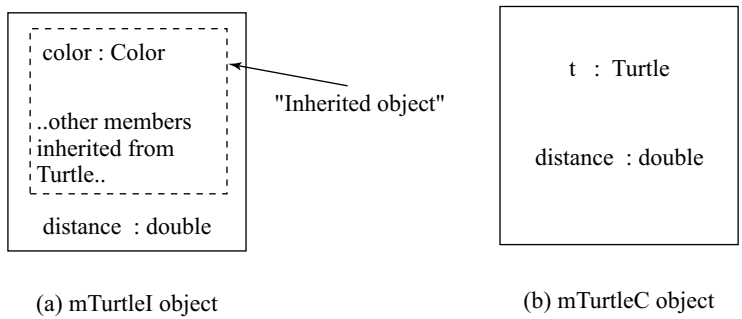


Fig. 25.1 *Contents of objects of class* `mTurtleI` *and class* `mTurtleC`

that are only meant for B. The body may also contain redefinitions of inherited member functions. For example, suppose the body contains a definition of f, which is an inherited member function, i.e. a function already defined in A. In such a case, the new definition is to be used with instances of B. The new definition is said to *override* the old definition, and will be used for objects of class B. The definition of f from A will continue to be used for objects of class A. Instances of class B can also use the old definition from A if necessary. Only, to do that, a slightly involved syntax is needed. Instead of just using the name f of the function, we must write A::f.

We have seen examples of all this in the definition of mTurtleI in Section 25.1.2. We defined a new member function distanceCovered, which is not present in the superclass Turtle, but is special to the class mTurtleI. We also redefined the member function forward present in Turtle. The new function changes the distance member appropriately, and also calls the function forward in class Turtle, using the syntax Turtle::forward. You can consider this function as being called on the inherited Turtle object inside mTurtleI.

Note that although objects of class B inherit all data members of A, their accessibility is limited. In fact, the accessibility of the inherited function members is also limited. We discuss the precise rules for this next.

25.2.1 Access Rules and protected Members

Suppose that m is a data or function member of A and b is an instance of B, a subclass of A, obtained using public inheritance. Then the manner in which the member m of instance b can be accessed is determined by the following rules.

❑ *Case 1:* m is a public member of A

In this case, we can consider m to be a public member of B as well. In other words, m can be accessed inside the definition of B, as well as outside the definition of B.

❑ *Case 2:* m is a private member of A

Then m cannot be accessed in the code that we write inside the definition of B. And of course it cannot be accessed outside. In other words, private members are accessible only inside the definition of the class in which the member was defined (and its friend classes). The subclass instances cannot directly access private members of the superclass. This is not to say that private members of the superclass are useless. There might well be a public or protected (see below) member function f of A which refers to m. Now, the code in B can refer to f, and hence it will indirectly refer to m.

❑ *Case 3:* m is a "protected" member of A

The notion of protected is as follows. If a member of a class A is designated as protected then it can be accessed only inside the definition of A or of its subclasses. In other words, a protected member is less accessible than a public member (which is accessible everywhere), and more accessible than a private member, (which is accessible only in the definition of A). Note that m is to be considered a protected member of B as well.

We illustrate the above rules using the following code snippet.

```
class A{
  int p;
protected:
  int q;
  int getp(){return p;}
public:
  int r;
  void init(){p=1; q=2; r=3;}
};

class B: public A{
  double s;
public:
  void print(){
    cout << p << endl;  // compiler error 1.  p is private.
    cout << q << ", "
         << r << ", "
         << getp() << endl;
  }
};

int main(){
  B b;
  b.init();
  cout << b.p         // compiler error 2.  p is private
       << b.q         // compiler error 3.  q is protected
       << b.r
       << b.getp()    // compiler error 4.  getp is protected.
       << endl;
  b.print();
}
```

If you compile this code, you will get the 4 compiler errors as marked. Compiler errors 1 and 2 are because `p` is private in `A`, and can hence not be accessed in the definition of `B`, or in `main`. Compiler errors 3 and 4 are because `q` and `getp` are protected in `A`, and hence cannot be used outside of the definition of `A` or of any subclass of `A`. Indeed you will see that protected members `q` and `getp()` can be used fine inside the definition of `B`. Further, the public members `init`, and `r` can be used if needed in both `B` as well as `main`.

Once the offending parts are removed, the code will compile fine. On execution, the print statement in `main` will print 3, and the statement `b.print()` will print 2,3,1.

Figure 25.2 shows the contents of the objects of classes `A` and `B`. Note that all data members from `A` are present in objects of class `B`, even if they might not be directly accessible.

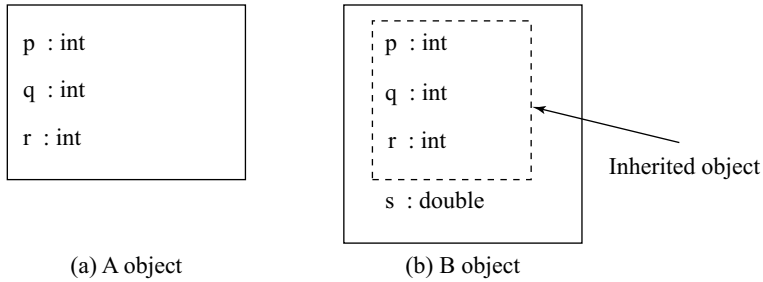


Fig. 25.2 *Contents of objects of class* `A` *and class* `B`

25.2.2 Constructors

Suppose class `B` is a subclass of class `A`. A constructor for `B` can to be defined using the following general form.

```
B(constructor-arguments) : call-to-constructor-of-A,
  initialization-list
  {
   body
  }
```

When you call the constructor for `B`, the `call-to-constructor-of-A` is first called, and this constructs the inherited object (of class `A`) contained inside the instance of `B` being created. The `initialization-list` has the form as in Section 18.1.5, and is used to initialize the new members of `B`. After that, `body` is executed. The part

```
: call-to-constructor-of-A
```

is optional. If it is omitted, the default constructor of `A` gets called. The `initialization-list` is also optional, and alternatively, the new members could be initialized inside `body`.

In Section 25.1, you saw an example in which the default constructor of `Turtle` got used for creating a `mTurtleI`. Suppose now that we had an alternate constructor for `Turtle` which took arguments `x,y` giving the initial position for the turtle. Then we could write an alternate constructor for `mTurtleI` as follows.

```
mTurtleI(double x, double y) : Turtle(x,y), distance(0) {}
```

With this constructor, the metered turtle would be created at position `(x,y)` on the screen, and the new member `distance` would be initialized to 0 using the initialization list. The body of the constructor would then be left empty.

25.2.3 Destructors

As before, suppose we have a class `B` which inherits from class `A`. Then the destructor for class `B` should be used to destroy the new data members introduced in `B` that were not present in `A`. The data members inherited from `A`? These would be destroyed by an implicit call that would get made to the destructor of `A` at the end of the execution of the call to the destructor of `B`. You should not explicitly make a call to the destructor of `A` from inside the destructor of `B`!

The general rule is: destruction happens automatically, in reverse order of creation. In the exercises you will experiment with code which will illustrate these ideas.

25.2.4 Other Operations on Subclass Objects

Besides constructors and destructors, one additional function is not inherited: the assignment operator. Unless you define an assignment operator, a default operator that does member by member copy will get defined for you. Note that the inherited object will be copied using its assignment operator.

Other operators and member functions will be inherited. You may of course override these definitions, as we did for `forward` in `mTurtleI`.

25.2.5 The Type of a Subclass Object

We have said that a class is a type, i.e. an object of class `A` has type `A`. Suppose `B` is a subclass of class `A`. Then a key idea in inheritance is: objects of class `B` can be considered to have type `B` as well as type `A`. This idea turns out to be quite useful.

The following analogy might be useful to understand this. Consider the category of flowers, in which we have subcategories such as roses, lotuses and so on. So if someone has demanded flowers, we can give roses. In other words, a specific rose object is useful as a rose as well as as a flower.

25.2.6 Assignments Mixing Superclass and Subclass Objects

If an object can be considered to be of the type of its superclass, we should allow an object of a subclass to be assigned to variable of the superclass. This is indeed possible. Note, however, that the subclass might have some additional data members. In such a case, the additional members are dropped, or *sliced* off, during the assignment.

You can also assign the address of a subclass object into a pointer variable of the superclass.

Figure 25.3 shows some examples. First, we have tha assignment `a = b`. Since `b` has an extra attribute `y`, during the assignment this will be sliced off, and only the attribute `x` will get copied. Thus, the first print statement will print 2, the value that got copied.

Next we have `aptr = &b`, which stores the address of the class `B` object `b` into the pointer variable `aptr` of type `A*`. As we said, this is allowed, since `A` is a superclass of `B`. The next statements use `aptr`. In the first, we access the member `x`, using the standard syntax `aptr->x`, and this will print 2 as expected. In the next statement, `aptr->f()`, we have invoked the member function `f`. Here there is some possible confusion: will this mean the `f` defined in `A` or the `f` defined in `B`? The default answer is that since `aptr` is of type pointer to `A`, the members from `A` will be used. Thus, this will print the message `"Calling f of A."`. If you are unhappy with this default, hold on till Section 25.3.

Note that in the above code, we cannot assign an object of the superclass into a variable of the subclass, i.e. write something like `b = a;`. The intuition behind this, going to our flower example, is as follows. Wherever a flower is expected, you can supply a rose; however, if a rose is expected, you cannot supply an arbitrary flower. Likewise, it is incorrect to write `bptr = &a` as well.

Finally, we can create references of the type superclass to objects of the subclass. This is done at the end of Figure 25.3. Indeed even in this case the function `f` in `A` will be invoked.

```
class A{
public:
  int x;
  A(){ x = 1; }
  void f(){cout <<"Calling f of A.\n";}
};
class B: public A{
public:
  int y;
  B(){ x = 2; y = 1;}
  void f(){cout <<"Calling f of B.\n";}
};

int main(){
  A a, *aptr;
  B b, *bptr;

  a = b;       // member y will be sliced off
               // member x will be copied.
  cout << a.x << endl;  // prints 2;

  aptr = &b;   // assigning subclass object to superclass pointer
  cout << aptr->x << endl; // prints 2;
  aptr->f();

  A& aref = b; // reference of type A to variable of type B.
  aref.f();
}
```

Fig. 25.3 *Assignments mixing subclass and superclass*

25.3 POLYMORPHISM AND VIRTUAL FUNCTIONS

Consider the code of Figure 25.3. As we discussed above, the call `aptr->f()` will cause the function `f` in class `A` to be used. But you might say: `aptr` *really* points to an object of type *B* so isnt it more useful if the `f` in `B` were used? You can make this happen by declaring `f` to be a *virtual* function. For this, you simply add the keyword `virtual` before the definition of `f` in `A`. Thus, the definition of `A` would have to be

```
class A{
public:
  int x;
  A(){ x = 1; }
  virtual void f(){cout <<"Calling f of A.\n";}
};
```

The keyword `virtual` says that the definition of `f` should not be treated as a unique, final definition. It is possible that `f` might be over-ridden in a subclass, and if so, that definition of `f` which is most appropriate (most derived!) for the object on which the call is made should be considered. When we call `aptr->f()`, the most appropriate definition for `f` is the one in `B`, since `aptr` actually points to an object of type `B`. So that definition gets used, and our code will now indeed print `"Calling f of B."`.

Note further that if `f` is virtual, its most derived version will get used if it is invoked on a reference as well. Thus, the last statement of Figure 25.3 will also cause `f` from `B` to be invoked.

Here is a more subtle example of the same idea.

```
class Flower{
public:
  void whoAmI(){ cout << name() << endl; }
  virtual string name(){ return "Flower"; }
};

class Rose: public Flower{
public:
  string name(){ return "Rose"; }
};

int main(){
  Flower a;
  Rose b;
  a.whoAmI();
  b.whoAmI();
}
```

Executing `a.whoAmI()` will clearly cause `"Flower"` to be printed out. More interesting is the execution of `b.whoAmI()`. What should it print? The call `b.whoAmI()` is to the inherited member function `whoAmI` in the superclass `Flower`. That function `whoAmI` calls `name`, but the question is which `name`. Will it be the `name` in `Flower` or in `Rose`? The answer turns out to be the `name` in `Rose`, because (a) the object on which `whoAmI` is called is of type `Rose`, and (b) `name` is virtual. Thus, the most derived definition of `name` appropriate for the object on which it is invoked will be used. Since the object on which it is invoked is of type `Rose`, the `name` from that class will be used. Thus, the last statement will print `"Rose"`. Note that had we not used `virtual`, both statements would have printed `"Flower"`.

In this example, the call `name()` inside the member function `whoAmI` is said to be *polymorphic*, because the same call will either cause the function in `Flower` to be called, or the function in `Rose` to be called, depending upon the actual type of the object on which it is invoked. Note that the actual type will only be known during execution.

Likewise the calls `aptr->f()` and `aref.f()` in Figure 25.3 would be polymorphic if `f` is declared virtual.

25.3.1 Virtual Destructor

Suppose `aptr` is of type `A*`, and points to some object. Suppose we wish to release the memory. So we write `delete aptr;`. This will call the destructor, but the question again is, which destructor? By default, the destructor of `A` will be called. However, if the object pointed to by `aptr` is of type `B`, which is a subclass of `A`, then clearly we should be calling the destructor for `B`. We can force this to happen by declaring the destructor of `A` to be `virtual`. Indeed, whenever we expect a class to be extended, it is a good idea to declare its destructor to be virtual.

Here is an example.

```
class A{
public:
  virtual ~A(){cout <<"~A.\n";}
};
class B: public A{
  int *z;
public:
  B(){z = new int;}
  ~B(){
    cout <<"~B.\n";
    delete z;
  }
};

int main(){
  A* aptr;
  aptr = new B;
  delete aptr;
}
```

If we do not declare the destructor of `A` to be virtual, then after the operation `delete aptr;` in the main program, the memory allocated for `z` will not be freed. However, since we have declared the destructor of `A` to be virtual, the destructor of `B` will be called when `delete aptr;` is executed, instead of the destructor of `A`. Thus the the operation `delete z;` will take place. Of course, as always the destructor of `A` will also be called, since our rule is that the destructor of the superclass will be called automatically after the destructor of the subclass finishes its execution. Do compile and execute this code, you will see from the message what is called. Remove the keyword `virtual` and execute again, you will see that only the destructor of `A` is called.

25.4 PROGRAM TO PRINT PAST TENSE

Suppose you wish to write a program that takes as input a verb from the English language, and prints out its past tense. Thus, given “play”, the program must print “played”, given “write”, the program must print “wrote”, and so on. A simple implementation would be to store every verb and its past tense as strings in memory. Given the verb, we can then print out the corresponding past tense.

But you will perhaps observe that for most verbs, the past tense is obtained simply by adding a suffix “ed”, as is the case for the verbs “play”, “walk”, “look”. We may consider these verbs to be *regular*.

Verbs such as “be”, “speak”, “eat” which do not follow this rule could be considered *irregular*. Thus, it makes sense to store the past tense form explicitly only for irregular verbs; for regular `verbs` we could simply attach “ed” when asked. This can be programmed quite nicely using inheritance.

We define a class `verb` that represents all verbs; it consists of subclasses `regular` and `irregular` respectively. The definition of `verb` contains information which is common to all verbs. The definition of `regular` adds in the extra information needed for regular verbs, and similarly the definition of `irregular`.

```
class verb{
protected:
  string root;
public:
  string getRoot(){return root;}
  virtual string past_tense(){return ""};
};
```

The member `root` will store the verb itself. We have defined the member function `past_tense` to be *virtual*. For now it returns the empty string. But this is not important, since we expect to override it in the subclasses.

The subclasses `regular` and `irregular` are as you might expect.

```
class regular : public verb{
public:
  regular(string rt){
    root = rt;
  }
  string past_tense(){return root + "ed";}
};

class irregular : public verb{
  string pt; // past tense of the verb
public:
  irregular(string rt, string p){
    root = rt;
    pt = p;
  }
  string past_tense(){return pt;}
};
```

Thus, to create an instance `v1` that represents the verb “play” we would just write

```
regular v1("play");
```

After this, if we wrote `v1.past_tense()`, we would get the string `"played"` as the result. Similarly,

```
irregular v2("be","was");
```

would create an instance `v2` to represent the verb "be". And, of course, `v2.past_tense()` would return "was".

We now see how the above definitions can be used to write a main program that returns the past tense of verbs. Clearly, we will need to somehow store the information about verbs. For this, we use a vector. We cannot have a single vector storing both regular and irregular verbs. However, we can define a vector of pointers to `verb` in which we can store pointers to `irregular` as well as `regular` objects. Thus, the program is as follows.

```
int main(){
  vector<verb*> V;
  V.push_back(new regular("watch"));
  V.push_back(new regular("play"));
  V.push_back(new irregular("go","went"));
  V.push_back(new irregular("be","was"));

  string query;
  while(cin >> query){
    size_t i;
    for(i=0; i<V.size(); i++)
      if (V[i]->getRoot() == query){
        cout << V[i]->past_tense() << endl;
        break;
      }
    if(i == V.size()) cout << "Not found.\n";
  }
}
```

We begin by creating the vector `V`. We then create instances of regular and irregular verbs on the heap, and store pointers to them in consecutive elements of `V`. Finally, we enter a loop in which we read in a `query` from the user, check if it is present in our vector `V`. If so, we print its past tense. Note that if the `for` loop ends with `i` taking the value `V.size()`, it must be the case that no entry in `V` had its `root` equal to the `query`. In this case, we print `"Not found."`. As you know, the while loop will terminate when `cin` ends, e.g. if the user types control-d.

A number of points are to be noted regarding the use of inheritance in this example.

1. Our need was to represent the category of verbs which consisted of mutually disjoint subcategories of irregular and regular verbs. This is a very standard situation in which inheritance can be used.
2. The vector `V` is polymorphic: it can contain pointers to objects of type `irregular` as well as of type `regular`. We can invoke the operation `past_tense` on objects pointed to by elements of `V`, without worrying about whether the objects are of type `regular` or `irregular`. Because `past_tense` is virtual, the correct code gets executed.

25.5 ABSTRACT CLASSES

You will note that we return the empty string in the `past_tense` function in `verb`. Returning an empty string does not make sense, but we did this because we expected that the `verb` class would

never be used directly; only its subclasses would be used in which the function would get overridden. This idea works, but it is not aesthetically pleasing that we should need to supply an implementation of `past_tense` in `verb` expecting fully well that it will not get used.

One possibility is to only declare the member function `past_tense` in `verb`, and not supply any implementation at all. Unfortunately, whenever an implementation is not supplied, the compiler expects to find it somewhere, in some other file perhaps. If such an implementation is not given the compiler or the linker will produce an error message.

What we need is a way to tell the compiler that we do not at all intend to supply an implementation of `past_tense` for the `verb` class. This is done by suffixing the phrase "`= 0`" following the declaration. Thus, we would write

```
class verb{
 ...
 public:
  virtual string past_tense() = 0;
 ...
}
```

Writing "`= 0`" following the declaration of a member function tells the compiler that we do not intend to at all supply an implementation for the function. You may think of 0 as representing the `NULL` pointer, and hence effectively indicating that there is no implementation.

There is an important consequence to assigning a function to 0. Suppose a class `A` contains a member function `f` assigned to 0. Then we cannot create an instance of `A`! This is because for that instance we would not know how to apply the function `f`. In C++, classes which cannot be instantiated are said to be *abstract*. Indeed, the only way of making a class abstract is to assign one of its member functions to 0.

So in our case, if we assign `past_tense` to 0 in `verb`, then the class `verb` would become abstract. We would not be able to create instances of it. But this is fine, we indeed would not like users to instantiate `verb`, instead we expect them to instantiate either `regular` or `irregular`.

25.6 MULTIPLE INHERITANCE

Sometimes, an object can be thought of as a specialization of not one, but *two* other objects. For example, we might have an `Automobile` class and a `SolarPoweredDevice` class.

```
class Automobile{
  double mileage;
};
class SolarPoweredDevice{
  double cellEfficiency;
};
```

Suppose we also need to represent solar-powered automobiles, they would need to have features of both the `Automobile` class as well as `SolarPoweredDevice` class. We can get this by constructing a class `SolarPoweredAutomobile` which inherits from both!

```
class SolarPoweredAutomobile : public Automobile,
                               public SolarPoweredDevice {};
```

Now instances of the `SolarPoweredAutomibile` class would have members `mileage` as well as `cellEfficiency`, as you might expect. Function members would also be inherited from all the superclasses, as many as there might be.

There are some obvious problems: what happens if the parent classes `P1`, `P2` of a class `C` both have a member with the same name `m`? In this case, the child class would get two copies of the member, and you would have to refer to the copies as `P1::m` and `P2::m`.

25.6.1 Diamond Inheritance

Suppose the parents `P1`, `P2` of some class `C` themselves inherit from a common base class `GP`, as shown in Figure 25.4. In this case, the class `C` will actually get two copies of the inherited object `GP`, corresponding to `P1` and `P2`.

This case, in which we derive a class `C` by inheriting from parent classes `P1`, `P2`, which in turn inherit from a single class `GP` is said to constitute *diamond inheritance*. This is because the pictorial representation of the inheritance has the diamond shape, Figure 25.4.

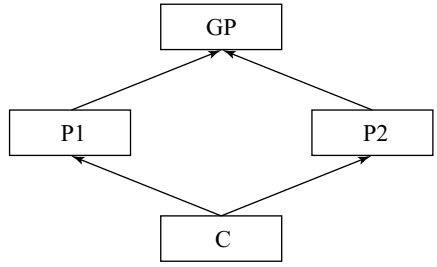


Fig. 25.4 *Diamond inheritance. Arrows go from child to parent.*

However, sometimes when we have diamond inheritance, we might want to have only one copy of the inherited object instead of one from each parent. It is possible to do this in C++ by specifying the derivation of `P1, P2` from `GP` as virtual. Thus, we would write

```
class GP{ double x; };
class P1: virtual public GP{};
class P2: virtual pubilc GP{};
class C: public P1, public P2{};
```

With this, the class `C` will contain only one copy of the inherited object `GP`. Note that this inherited object will be initialized directly by calling its constructor. Thus, if the initialization list of `P1` or `P2` contains a call to the constructor of GP, then those calls will be ignored.

25.7 TYPES OF INHERITANCE

So far we have only discussed `public` inheritance. C++ allows other kinds of inheritance also, namely `protected` inheritance and `private` inheritance.

If a class `B` inherits from a class `A` using `protected` inheritance, then the public and protected members of `A` become protected members of `B`.

If a class `B` inherits from a class `A` using `private` inheritance, then the public and protected members of `A` become private members of `B`.

As you can see, protected and private inheritance progressively restrict the accessibility of the members inherited into the derived class. These kinds of inheritance appear to be used much less in practice. So we will not discuss them any further.

25.8 REMARKS

Inheritance is a powerful idea. However, like any powerful idea, it needs to be used with care.

An informal rule of thumb is as follows. Suppose A, B are entities which you wish to represent using classes `A,B`. If entities of type B are specialized versions of type A, then inherit class `B` from class `A`. Informally, you may ask, do the entities B, A have an "is-a" relationship? Clearly, a rose is a flower, so the entities rose, flower have an "is-a" relationship. So in this case, use inheritance. If two entities have a "has-a" relationship, then better use composition. For example, flowers have petals, so the entities flower, petal have a "has-a" relationship. It is best to model this relationship by composition, i.e. by making a petal a member inside a flower.

An important advantage of inheritance is polymorphism. As we saw in Section 25.4, we might have several subclasses of a base class. We can conveniently store (pointers to) instances of the subclasses in a vector (or some other collection), for this we can consider them to be members of the base class. But we can invoke functions on the objects, and if the functions are virtual, we get the benefit of considering them to be members of the subclasses too. This came in handy when writing the program to display past tense of verbs. And we will see more examples of this in the next chapter.

EXERCISES

1. Suppose you have a class `V` defined as

```
class V{
protected:
  double x,y,z;
public
  V(double p, double q, double r){ x=p; y=q; z=r; }
}
```

   Define a class `W` that inherits from `V` and has a member function `dot` which computes the dot product of two vectors. Thus, given `V` type objects `v,w`, then the dot product is `v.x * w.x + v.y * w.y + v.z * w.z`. Be careful that you only use the constructor provided in `V`.

2. Define a class `realTurtle` such that `realTurtle` objects move with some specifiable speed when they move. They should also turn slowly.

3. What do you think will happen when you execute the program given below? Run it and check if you are right.

```
class A{
```

```
public:
  A(){cout << "Constructor(A).\n";}
  ~A(){cout << "Destructor(A).\n";}
};

class B: public A{
public:
  B(){cout << "Constructor(B).\n";}
  ~B(){cout << "Destructor(B).\n";}
};

class C: public B{
public:
  C(){cout << "Constructor(C).\n";}
  ~C(){cout << "Destructor(C).\n";}
};

int main(){
  C c;
}
```

4. What will the following code print?

```
struct A{
  virtual int f(){return 1;}
  int g(){return 2;}
};

struct B : public A{
  int f(){return 3;}
  int g(){return 4;}
}

A* aptr;
aptr = new B;
cout << aptr->f() <<' '<< aptr->g() << endl;
```

5. Write the past-tense generation program using just two classes, a `verb` class and an `irregular` class. A regular verb should be created as an instance of `verb`, and an irregular as an instance of `irregular`.

6. You might note that the past tense of several verbs ending in "e" is obtained by adding "d", e.g. recite, advise. Add an extra subclass to the verb class to store the past tense of such verbs compactly.

CHAPTER 26

Inheritance-based Design

Inheritance is often useful in designing large complex programs. Its first advantage we have already discussed: inheritance is convenient in representing categories and subcategories. But there are some related advantages which have to do with the management of the program development process. We will discuss these next, and then in the rest of the chapter we will see some examples.

There are many approaches to designing a large program. A classical approach requires that we first make a complete study of the program requirements, and only thereafter start writing the program. More modern approaches instead acknowledge/allow for the possibility that requirements may not be understood unless one has built a version of the program. Also, if a program works beautifully, users will inevitably ask that it be enhanced with more features. In any case, programs will have a long lifetime in which the requirements will evolve. So the modern approaches stress the need to design programs such that it is easy to change them. As we have discussed, the whole point of inheritance is to build new classes out of old, and this idea will surely come in useful when requirements change.

Even if the requirements are well understood and fixed (at least for that time in the program development process), designing a large program is tricky. It greatly helps if the program can be designed as a collection of mostly independent functions or classes which interact with each other in a limited, clearly defined manner. Partitioning into nearly independent parts is helpful in understanding the behaviour of the program and also checking that it is correct: we can consider testing the parts separately for example. But it also has another advantage: different programmers can work on the different parts simultaneously. As we will see, inheritance based designs have much to offer in this regard also.

Another modern programming trend is the use of *components*. Most likely, to write professional programs you will not use bare C++ (or any other programming language), but will start from a collection of functions and classes which have been already built by others (for use in other, similar projects). You will adapt the classes for your use, and as we have seen, this adaptation is natural with inheritance.

In this chapter, we will mainly see two case studies. First, we revisit our formula drawing program. We will rewrite it using inheritance. It will turn out that this way of writing makes it easier to maintain and extend. Next we will discuss the design of the graphics in Simplecpp. Inheritance plays a substantial role in its design. Finally, we will see the `Composite` class, which will enable you to define new graphical objects by composing the simple graphics objects we know so far.

26.1 FORMULA-DRAWING REVISITED

Consider the formula-drawing problem from Section 24.2: given a mathematical formula, draw it on the graphics canvas in a nice manner. In this section, we will see how the program can be written using inheritance. A benefit of this will be that it will be easy to extend the program to include new operators. To illustrate this, we will show how to extend the program (which in the intial implementation only allows the + and / operators as in Section 24.2) to include the exponentiation operator.

For simplicity, we ignore the problem of accepting input from the keyboard: we will assume that the formula to be drawn is given as a part of the program, e.g. to draw $1 + \frac{2}{\frac{451}{5}+35}$ we will construct it in our program by writing something like

```
Node e('+', new Node("1"),
       new Node('/', new Node("2"),
           new Node('+', new Node('/',new Node("451"),new Node("5")),
               new Node("35"))));
```

as discussed in Section 24.2.6. Then we can call `e.setSizes()` and so on.

26.1.1 Basic Design

The use of inheritance is natural if the entities we want to represent belong to a category and subcategories thereof. We can think of mathematical formulae as constituting a category. But can we divide the category of mathematical formulae into subcategories in a useful manner?

From the purpose of drawing the formula on the canvas, the important consideration is what is the *last* operation in the formula. For example in the formula $\frac{a}{b+c}$, the last operation to be performed is division, and hence we will designate this formula to be of the subcategory *ratio*. In the formula $b + c$, the last (and only) operation to be performed is addition, so we will designate it to be of the subcategory *sum*. There could also be categories such as products or differences. Another category is *literal*, which will contain primitive formulae, i.e. in which there is no last operation. Note that a formula such as $\frac{a}{b+c}$, of category ratio, will contain inside it other formulae: a formula a which is a literal and another formula $b + c$ which is a sum.

As discussed earlier, we will have classes and subclasses according to categories and subcategories. Thus, we will have a class `Formula` of which `Sum` and `Ratio` will be subclasses. And we will have a subclass `Literal` to represent primitive formulae. We could also have subclasses `Product` and `Difference`, but we will omit these for brevity.

What members appear in each class is decided using the following rule: if a certain member is needed in all subclasses of a class, then it should be defined in the class rather than the subclasses. Thus, for example, we expect all formulae to have a width, height, ascent, and descent. So these members will be placed in the `Formula` class. Likewise we expect to have member functions `draw` and `setSize` (as in Section 24.2.5) for all formulae. These also we will place in `Formula`. Here is the definition of the class `Formula`.

```
class Formula{
protected:
  double width, height, ascent, descent;
public:
  virtual void setSizes()=0;
  virtual void draw(double x, double y)=0;
```

```
  double getWidth(){return width;}
  double getHeight(){return height;}
  double getDescent(){return descent;}
  double getAscent(){return ascent;}
};
```

We do not expect `Formula` to be instantiated, so we have declared some of its member functions to be pure virtual. We have added the accessor functions for width etc. since they might be needed outside the class definition.

We can make `Sum` and `Ratio` be subclasses of `Formula`, but note that there is some similarity in sums and ratios: they both take two operands. So it is a good idea to have an intermediate class of formulae in which the last operation takes two operands.

```
class Formula2: public Formula{
protected:
  Formula *L, *R;
};
```

Next we create the `Sum` class. It will provide implementations for `draw` and `setSizes`—the logic of this will follow that of Section 24.2.4. Specifically, we will adapt the code for case `'+'` from Section 24.2.5. We will make `Sum` a subclass of `Formula2`, that way it will inherit members `L, R`.

```
class Sum: public Formula2{
public:
  Sum(Formula* l, Formula* r){ L =l; R=r;}
  void setSizes(){
    L->setSizes();
    R->setSizes();
    width = L->getWidth() + textWidth(" + ") + R->getWidth();
    descent = max(L->getDescent(), R->getDescent());
    ascent = max(L->getAscent(), R->getAscent());
    height = ascent + descent;
  }
  void draw(double x, double y){
    L->draw(x, y + ascent - L->getAscent());
    R->draw(x + L->getWidth() + textWidth(" + "),
            y + ascent - R->getAscent());
    Text(x + L->getWidth() + textWidth(" + ")/2, y + ascent,
         string(" + ")).imprint(); // draw the '+' symbol
  }
};
```

We will also make `Ratio` be a subclass of `Formula2`. The code for ratios is also based on case `'/'` of Section 24.2.5.

```
class Ratio: public Formula2{
  static const int h_bar = 10;
public:
  Ratio(Formula* l, Formula* r){ L = l; R = r;}
```

```
  void setSizes(){
    L->setSizes();
    R->setSizes();
    width = max(L->getWidth(), R->getWidth());
    ascent = h_bar/2 + L->getHeight();
    descent = h_bar/2 + R->getHeight();
    height = ascent + descent;
  }
  void draw(double x, double y){
    L->draw(x + width/2 - L->getWidth()/2, y);
    R->draw(x + width/2 - R->getWidth()/2, y
    + L->getHeight() + h_bar);
    Line(x, y + ascent, x + width, y + ascent).imprint();
     // horizontal bar
  }
};
```

Finally, we need a class to represent primitive formulae, or literals. Literals only contain a value, but no subformulae. Thus, they must inherit from `Formula` and not `Formula2`.

```
class Literal : public Formula{
  string value;
public:
  Literal(string v){value=v;}
  void setSizes(){
    width = textWidth(value);
    height = textHeight(); ascent = descent = height/2;
  }
  void draw(double x, double y){
    Text(x+width/2,y+height/2,value).imprint();
  }
};
```

This code is based on case `'P'` of Section 24.2.5.

Note that our hierarchy has resulted in good use of memory. In each object, we have just the members that we need. In `Literal` the members `L, R` are not present; while `value` is not present in `Sum` or `Ratio`. Also note the benefit of having the class `Formula2`: we have only one declaration of `L, R` which is then inherited into `Ratio` and `Sum`.

We can now give a simple main program which can use the above definitions to render the formula $1 + \frac{2}{\frac{451}{5}+35}$.

```
int main(){
  initCanvas("Formula drawing");
  Sum e(new Literal("1"),
        new Ratio(new Literal("2"),
                  new Sum(new Ratio(new Literal("451"),new Literal("5")),
                          new Literal("35"))));
```

```
  e.setSizes();
  e.draw(200,200);

  getClick();
}
```

26.1.2 Comparison of the Two Approaches

One benefit of the new approach we have already mentioned: it saves memory. In the program of Section 24.2, we had `Node` objects instead of `Formula` objects. A `Formula` object either has members `L, R`, or it has the member `value`. On the other hand, each `Node` object has all members `L, R, value, op`, whether or not they were needed. On the downside, it might seem that the inheritance based approach is more verbose than the approach of Section 24.2. This is true, but the verbosity has bought us many things.

A key improvement is that we have partitioned the program into manageable pieces. The code of Section 24.2 had just one class. All the complexity was placed into that class. In contrast, in the new code, different concerns are separated into different classes. For example, the class `Formula2` only models the fact that formulae can have two operands, nothing more. Or the class `Sum` holds all information about layout of sums, and nothing more. We can place each class into its header and implementation files, and the main program into a separate file. This way, if we wish to change something regarding a certain issue (e.g. adjust the spacing between the numerator and the denominator in a ratio) we know that we will likely modify only one small file.

Another important benefit arises when we consider adding new functionality to the program. Suppose we want to implement layouts of exponential expressions. As you will see, we can do this without touching any of our old files. The key benefit of this strategy is: we can be sure that when we add exponential expressions, *there isnt even a remote chance of damaging the old working code*. Programmers deservedly tend to be paranoid about their code, and this kind of reassurance is useful. Notice that if the old code was written by one programmer, and the new one by another, then it is very convenient if one programmer's code is not touched by another. This way there is clarity about who was responsible for what.

26.1.3 Adding the Exponentiation Operator

We will add a class `Exp` that will represent formulae in which the last operator is exponentiation. This will be a subclass of `Formula2` since exponentiation has 2 operands: the base and the exponent. We will use `L` to point to the base and `R` to point to the exponent.

```
class Exp : public Formula2{
public:
  Exp(Formula* l, Formula* r){ L = l; R = r;}
  void setSizes(){
    L->setSizes();
    R->setSizes();
    width = L->getWidth() + R->getWidth();
    descent = L->getDescent();
    ascent = L->getAscent() + R->getHeight() - textHeight()/3;
```

```
    height = ascent + descent;
  }
  void draw(double x, double y){
    L->draw(x, y + R->getHeight() - textHeight()/3);
    R->draw(x + L->getWidth(), y);
  }
};
```

The basic idea is to layout the exponent somewhat above and to the right of the base. Ideally we should use a smaller font for drawing the exponent. However, Simplecpp only has fonts of a single size, so we will not worry about this. The detailed expressions which decide how to position the base and the exponent are obtained in the manner of Section 24.2.4, and are left for you to figure out. Here is a main program that uses this to produce a layout for $\frac{x^2+y}{x+y^{23}}$.

```
int main(){
  initCanvas("Formula drawing");

  Ratio f(new Sum(new Exp(new Literal("x"), new Literal("2")),
                  new Literal("y")),
          new Sum(new Literal("x"),
                  new Exp(new Literal("y"), new Literal("23"))));

  f.setSizes();
  f.draw(200,300);

  getClick();
}
```

Figure 26.1 shows the result.

The key point to appreciate is that this new code can be developed independently, in a new file, without needing to understand the rest of the code. The developer of `Exp` would only need the header files of classes `Formula2` and `Formula`. If we had used the coding approach of Section 24.2, we would need to have and modify the member functions in the old code.

26.1.4 Reading in Formulae

To read a formula from the keyboard, the program must know about all acceptable types of formulae. So if you wish to extend your program by adding new operators, the code for reading formulae will have to be modified; you cannot simply include new code and not have to modify old code.

But as we showed above, the code for processing formulae can be written separately for each formula type, and it should be.

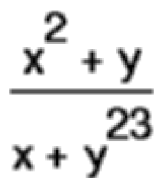


Fig. 26.1 *Formula with exponentiation*

26.2 THE SIMPLECPP GRAPHICS SYSTEM

We will discuss the role played by inheritance in the design of the Simplecpp graphics system. We will not discuss the entire system here, but only some relevant portions of it.

Briefly stated, the core specification of the system is: allow the user to create and manipulate graphical objects on the screen. This statement is very vague, of course. What does it mean to *manipulate* objects? As you know, in Simplecpp, manipulate means move, rotate, scale, etc. There are also other questions: what kind of primitives is the user to be given? Will the user need to specify how each object appears in each *frame* (like the picture frames in a movie) or will the user only state the incremental changes, e.g. move object x, which means the other objects remain unchanged? As you know, we have opted for the latter. And then of course there is the question of what kinds of objects we can have. As you know, Simplecpp supports the following kinds of graphical objects: circles, lines, rectangles, polygons, turtles and text.

So in the rest of this section, we consider the problem of creating and manipulating the objects given above, in the manner described above. We will not discuss issues such as the pens associated with each object, or graphical input, as in the `getClick` command. Clearly, such a system must have the following capabilities:

1. It must be able to keep track of the objects created by the user so far.
2. It should be able to display the objects on the screen.
3. It should be able to manipulate the objects as requested, e.g. move an object.

Clearly, we need to record the information about what graphics objects have been created. This information will be of two types, information having to do with the shape of the object, and other information such as its colour, position, orientation, etc. Using this information, we can display the objects. For this, Simplecpp calls the graphics commands provided by the underlying operating system. For example, the underlying system will provide commands that enable you to draw lines at specified positions—this will be invoked whenever needed. If the user decides to change the position of a line, then the information stored about the line must be suitably changed. To complete this very high level description we need to answer one more question: when should the objects be displayed? The simplest answer to this is: whenever the user changes the state of any object, we must clear the entire display and display all objects again.

Inheritance is useful for dealing with many of the issues described above. Clearly, we can consider all graphical objects to comprise a category. Objects of each shape will comprise a subcategory. Thus, we can have a class for all objects, and subclasses for the subcategories. The heirarchy formed by our classes is shown in Figure 26.2. The class associated with the category of all objects is called `Sprite`, in honour of the Scratch programming environment (`scratch.mit.edu`), where the name is used for a similar concept.

The hierarchy facilitates storing of information as follows. In the `Sprite` class, we keep all attributes that all graphics objects will possess. These are attributes such as position, orientation, scale, colour. So these attributes become data members in the `Sprite` class.

The classes `Circle, Line, Polygon` etc. will contain the shape related attributes (in addition to all the attributes inherited from `Sprite`). For example, `Circle` contains a data member called `radius` which holds the radius of the circle being drawn. The `Polygon` class contains an array which holds the coordinates of the vertices of the polygon. Figure 26.3 shows possible contents of the

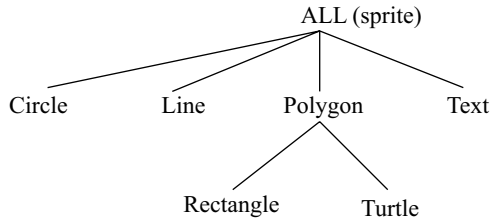


Fig. 26.2 *Hierarchy of graphics object categories*

```
class Sprite{
protected:
  double x,y;           // position on the canvas
  double orientation;   // angle in radians made with the x axis
  double scale;         // scaling factor
  Color fill_color;     // Color is a data type in the X windows
                           Xlib package.
  bool fill;            // whether to fill or not
  ...
 public:
  Sprite();
  Sprite(double x, double y);
  ...
  void move(double dx, double dy);
  ...
  virtual void paint()=0;
  ...
}

class Circle : public Sprite{
 private:
  double radius;
 public:
  Circle();
  Circle(double x, double y, double radius=10);
  virtual void paint();
};
```

Fig. 26.3 *Possible definitions of* `Sprite` *and* `Circle`

`Circle` and `Sprite` classes. The actual implementation is different, and you can see it in the code (Appendix A).

In addition, we must also consider function members. Suppose we wish to move an object. This requires two actions: (a) recording that the object has indeed been moved, and updating its position accordingly, (b) redrawing the object on the screen. Clearly, action (a) can be performed independent of the shape of the object, whereas (b) requires the shape information. Thus, in our implementation,

action (b) is implemented by a `paint` member function in each shape class. Member functions such as `move` are defined in the `Sprite` class. The `move` function performs action (a) using the attributes available in the `Sprite` class. It then signals that all objects need to be redrawn.

The redrawing works as follows. Essentially, Simplecpp maintains a vector that holds pointers to all objects active at the current instant. Suppose the vector is named `ActiveSprites`, then its declaration would be

```
vector<Sprite*> ActiveSprites;
```

Because the elements of `ActiveSprites` have type `Sprite*`, they can hold pointers to any graphical object. When the objects are to be drawn, we simply iterate over the queue and execute the `paint` member function of the object.

```
for(int i=0; i < ActiveSprites.size(); i++){
  ActiveSprites[i]->paint();
}
```

The `paint` member function is virtual, and so the `paint` member function in the class of the object is used. This is similar to the way we used a vector of `Verb*` to store `regular` and `irregular` objects and invoked the `past_tense` virtual function on them in Section 25.4.

In summary, inheritance gives us three main benefits. The inheritance heirarchy helps in organizing our data: the `Sprite` class stores attributes that are possessed by all graphics objects, and the other (shape) classes store attributes related to the shape. Also because of polymorphism and virtual functions, we can store pointers to different types of graphical objects (but only subtypes of `Sprite`) in a single vector, and iterate over the vector. Finally, we can add new shapes easily: we simply define a new shape class which is a subclass of `Sprite`, without having to modify any existing code.

We have somewhat simplified the description of Simplecpp graphics in order to explain the use of inheritance. The actual system is more complex.

26.3 COMPOSITE GRAPHICS OBJECTS

We present the `Composite` class of Simplecpp. This will help you to group many graphics objects into a single composite object.

Suppose you want to draw cars on the screen. It would be nice if you could design a class `Car` which you could then instantiate to make many cars. A car is a complex object: it cannot be drawn nicely using just a single polygon, or a single circle. It will require several simple objects that Simplecpp provides. These simple objects will have to be grouped together, and often be manipulated together, e.g. if we want the car to move, we really mean to move all its constituent parts. The class `Composite` which we discuss next, will allow you to group together objects. We can then define a `Car` class by inheriting from the `Composite` class.

Our `Composite` class primarily serves as a "container" to hold other graphics objects. It has a frame of reference, relative to which the contained objects are specified. The `Composite` class has been defined as a subclass of the `Sprite` class. Thus, it inherits member functions such as move, forward, rotate from the Sprite class. The `Composite` class is designed so that the member functions will cause the contained objects to respond appropriately, i.e. when you move a `Composite`, everything inside gets moved. However, you can override these member functions if you wish. For example, suppose your composite object consists of the body of a car and its wheels. When you

call `forward` on this, by default everything will go forward. You might want the wheels to rotate in addition to moving forward. This can be accomplished by overridding. You can also additionally define your own new member functions which do new things. For example, the car might have a light on the top and there could be a member function which causes the light to change colour from white to yellow (suggesting it is switched on). This could be done using a new member function.

Using the `Composite` class is fairly easy. There are only two important ideas to be understood: the notion of ownership, and the `Composite` class constructor.

26.3.1 Ownership

A detail we have hidden from you so far is: every graphics object has an "owner". When we say that object `X` owns object `Y`, we merely mean that object `Y` is specified relative to the coordinate frame of object `X`. For the objects you have been creating so far, the owner was the canvas: the objects were drawn in the coordinate frame of the canvas. When an object is created as a part of a composite object, it must be drawn relative to the frame of the composite object, and hence must be owned by the composite object. Thus, an important step in defining a composite object is to declare it to be the owner of the contained objects.

To do this, the constructor of every graphical object is provided with an optional argument named `owner`. This argument takes value `NULL` by default which Simplecpp interprets to mean the canvas. Thus, so far, we did not tell you about this argument, and you did not specify a value for it. Hence Simplecpp made the canvas the owner of all the objects you created. If you want to indicate a different owner, you instead pass a pointer to that owner. In the present case, we want a composite object to be the owner of the contained object. Thus we pass a pointer to the composite object while creating the contained objects.

26.3.2 The `Composite` Class Constructor

The `Composite` class constructor has the following signature.

```
Composite(double x, double y, Composite* owner=NULL)
```

Here, the last argument `owner` gives the owner of the composite object being defined, and `x,y` give the coordinates of the composite object in the frame of its owner. As mentioned earlier, if you do not specify this argument, it is taken as `NULL`, indicating that the canvas is the owner. The `owner` argument must be specified if this composite object is itself a part of another composite object. This kind of containment is allowed and we will see an example shortly.

The `Composite` class is essentially abstract; you do not instantiate it directly. You inherit from it to define composite classes that you need, and then instantiate those classes.

26.4 A `Car` CLASS

Our car will consist of a polygonal body, and two wheels. We will give the wheels some personality by adding in spokes. So we will model a car as a composite object, consisting of the body and the wheels. But note that a wheel itself contains a circle representing the rim, and lines representing the spokes. So the wheel will itself have to be represented as a composite object. Note that we allow one composite object (e.g. a car) to contain other ordinary objects (e.g. body) or other composite objects (e.g. wheels).

We begin by defining a class for wheels.

```
const double RADIUS = 50;
class Wheel : public Composite{
  Circle *rim;
  Line *spoke[10];
public:
  Wheel(double x, double y, Composite* owner=NULL) :
   Composite(x,y,owner){
    rim = new Circle(0,0,RADIUS,this);
    for(int i=0; i<10; i++){
      spoke[i] = new Line(0, 0, RADIUS*cos(i*PI/5),
       RADIUS*sin(i*PI/5), this);
    }
  }
};
```

There are two private data members. The member `rim` which is defined as a pointer to the `Circle` object which represents the rim of the wheel. Likewise, `spoke` is an array of pointers to each spoke of the wheel. The objects themselves are created in the constructor. This is a very common idiom for defining composite objects.

The constructor customarily takes as argument a pointer to the owner of the composite object itself, and the position of the composite object in the frame of the owner. It is customary to assign a default value `NULL` for the `owner` parameter, as discussed earlier. The initialization list `Composite(x,y,owner)` merely forwards these arguments so that the core composite object is created at the required coordinate and gets the specified owner. Inside the constructor, we create the sub-objects. So we create the circle representing the rim, and as you can see we have given it an extra argument `this`, so that the `Wheel` object becomes the owner of the rim. Likewise, we create lines at different inclinations to represent the spokes, and even here the extra argument `this` causes the lines to be owned by the `Wheel` object.

The `Car` class can be put together by using `Wheel` instances as parts.

```
class Car : public Composite{
  Polygon* body;
  Wheel *w1, *w2;
public:
  Car(double x, double y, Color c, Composite* owner=NULL)
    : Composite(x,y,owner){
    double bodyV[9][2]={{-150,0}, {-150,-100}, {-100,-100},
                        {-75,-200}, {50,-200}, {100,-100},
                        {150,-100}, {150,0},
    {-150,0}};
    body = new Polygon(0,0, bodyV, 9, this);
    body->setColor(c);
    body->setFill();
    w1 = new Wheel(-90,0,this);
    w2 = new Wheel(90,0,this);
  }
```

```
  void forward(double dx){
    Composite::forward(dx);       // superclass forward function
    w1->rotate(dx/(RADIUS*getScale())); // angle = dx/current-RADIUS
    w2->rotate(dx/(RADIUS*getScale()));
  }
};
```

As will be seen, the private members are the pointers to the body and the two wheels. In the constructor, the body is created as a polygon. We have provided a parameter in the constructor which can be used to give a colour to the body. Finally, the wheels are created. For all three parts, the last argument is set to `this`, because of which the parts become owned by the `Car` object, as we want them to be.

The definition also shows the `forward` function being overridden. As discussed, we want the car to move forward, which is accomplished by calling the forward function of the superclass. But we also want the wheels to turn; this is accomplished by rotating them. Clearly, if the car moves forward by an amount `dx`, then the wheels must rotate by `dx/r` radians, where `r` is the current radius of the wheels, i.e. the actual radius multiplied by the current scale factor.

Finally, here is a main program that might use the above definitions.

```
int main(){
  initCanvas("Car",800,800);
  Car c(200,300,COLOR("blue")), d(200,600,COLOR("red"));
  d.scale(0.5);

  getClick();

  for(int i=0; i<400; i++){
    beginFrame();
    c.forward(1); d.forward(1);
    endFrame();
  }
  getClick();
}
```

The main program creates two cars, one blue and another red. The red car is then scaled to half its size. This causes all the components of the car to shrink—this is handled automatically by the code in the `Composite` class.

Finally, we move the cars forward. When you execute this, the car wheels should appear to roll on the ground. Further, the wheels of the smaller car should appear to have twice as many rotations per unit time because the smaller wheel has half the radius but is travelling the same distance as the larger wheel.

Finally, note that we have used `beginFrame` and `endFrame` (Section 20.2). If you remove these commands, you will see that the different parts of the car move, but not together!

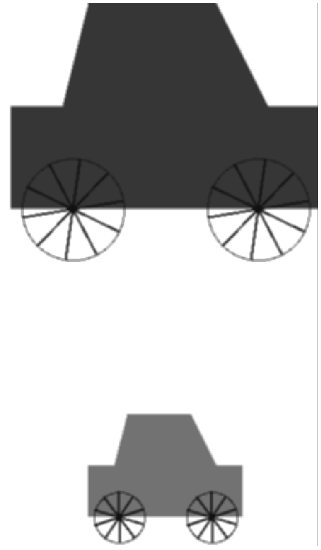


Fig. 26.4 *Cars created by the program*

26.5 CONCLUDING REMARKS

A number of ideas were discussed in this chapter.

The first important idea was that we should consider using inheritance whenever we wish to represent entities belonging to categories and subcategories. It is often tricky to decide what the precise subcategories should be. In our formula drawing example, it was perhaps clear that categories like formula, sum, ratio, were needed. It might have been less clear that a category like `Formula2` was also needed. But the operative principle here is: ensure that each member is defined in only one class. This should lead us to having a class `Formula2`. Without it, we would need to duplicate `L, R` in `Sum` and `Ratio`. The other possibiity is to put `L, R` in `Formula` itself, but then these members would be present in classes like `Literal` as well, where they are not needed.

A case could be made for having even more subclasses. For example, one important idea in object oriented design is the notion of an *interface*. The term *interface* when formally used, typically denotes a class without any data members. It only contains virtual functions. An interface thus describes the functionality required from the class, without making any suggestions whatsoever about how the functionality is to be implemented. So we might have a `Formula` class without any data members. These might get introduced in a subclass. This would give more flexibility in future extension. Another possibility is to note that different types of formulae might share the same layout strategy. For example, products, differences and sums, are all laid out horizontally, with only the operator between the operands being different. Thus, we might have a subclass `Formula2h` which would specialize to `Sum, Product, Difference` but not `Ratio`. `Formula2h` would have the `setSizes` and `draw` member functions which would be inherited by the `Sum, Product` and `Difference`. The subclasses of `Formula2h` would merely have to define what operator to use in the layout.

It should be noted that the formula drawing example is a bit confusing because there are really two hierarchies in it. The first is the class hierarchy: `Formula` functionally specializes to `Formula2` which further specializes to `Sum, Ratio`, and so on. But there is also a containment hierarchy. A formula such as $\frac{a}{b+c}$ is a `Ratio` formula which contains a `Sum` formula. In many applications, both

hierarchies are present, and it is important not to confuse between the two. An important idea regarding containment hierarchies is that operations on the container are often implemented recursively, i.e. drawing a sum is accomplished by drawing the formulae contained in the sum. You will see examples of such recursion in many applications.

The use of polymorphism should also be noted. When we said that a `Sum` contains two operands which are themselves formulae, we can indicate the type of the operands as `Formula`, without having to further specify whether they are `Sum`s or `Ratio`s or whatever. Another example of this appeared in Simplecpp: we had to put various graphics objects into a queue, and within the queue they would be viewed just as `Sprite`s. However, when we call the `paint` member function, the correct code would get executed because `paint` is virtual.

It should be noted that the implementation of `Composite` in Simplecpp is somewhat preliminary. It will not properly implement movement of pens associated with objects contained inside a `Composite` object. Copying `Composite` objects will also not work correctly; however it should be fine if you pass a `Composite` object by reference. However, even with these shortcomings you should be able to use `Composite` to write interesting programs.

EXERCISES

1. To the program of Section 26.1, add the capability of drawing summation formulae, i.e. formulae

$$\sum_{A}^{B} C$$

where A, B, C can themselves be formulae. Ideally you should draw the expressions A, B using a smaller font. However, Simplecpp only has fonts of one size so do not worry about that.

2. Modify the formula-drawing program developed in this chapter so as to include the additional classes discussed in Section 26.5.

3. Write a program using which non-programmers can write simple animations as follows. The input to the program could be something like the following sequence of commands.

```
Circle 100 200 5
Rectangle 200 300 40 50
Circle 100 200 15
Move 0 50 50
Left 1 30
Move 2 70 -70
Wait 0.5
...
```

In this, the first 3 lines defined 3 objects. As you might guess, the numbers following the shape names are the arguments for the corresponding constructors. For the rest of the sequence, the constructed objects respectively get numbered 0, 1, 2 (or till as many objects as we have defined). In the rest of the command sequence, a graphical object will be referred to by its number. Thus,

the command *Move 0 50 50* causes object 0 (the first circle) to be displaced by 50, 50 along x and y directions. Likewise `Left 1 30` causes the rectangle to be rotated left by 30 degrees. You may also define commands to wait for specified amount of time.
Write the program. Note that `Sprite` objects cannot be stored in a `vector`. However, you can create the `Sprite` on the heap and store a pointer to it in a vector.

4. Add lights to the car of Section 26.4. They should turn on, i.e. change colour, when the car is moving.

5. Write a program in which a stick figure of a human being walks across the screen.

CHAPTER 27

Discrete-event Simulation

We have already discussed the general notion of simulation: given the current state and laws of evolution of a system, predict the state of the system at some later date. In Chapter 19, we considered the simulation of heavenly bodies, as might be required in astronomy. However, simulation is very useful also for more mundane, down to earth systems. A very common use of simulation is to understand whether a facility such as a restaurant or a train station or airport has enough resources such as tables or platforms or runways to satisfactorily serve the customers or travellers that might arrive into it.

As a concrete example, suppose we want to decide how many dining tables we should put in a restaurant. If we put too few tables, we will not be able to accommodate all customers who might want to eat in our restaurant. On the other hand, each table we put in has a cost. So we might want to determine, for each T where T is the number of tables we put, what our revenue is likely to be. Knowing the revenue and the cost of putting up tables, we will be able to choose the right value of T. To do this analysis, we of course need to know something about how many customers want to eat in our restaurant, and when. This can be predicted only statistically. We will assume that we are given p, the probability that a customer arrives in any given minute of the "busy period" for restaurants, say 7 p.m. to 10 p.m. Ideally, we should consider not single customers but a party consisting of several customers, and the possibility that a customer party might need more than one table. However, for simplicity we will assume that customers arrive individually and are seated individually at separate tables. On arrival, a customer occupies the table for some time during which he eats, and then leaves. Suppose that we are also given a function $e(t)$, that gives the probability that a customer eats for t minutes. For simplicity, suppose that the revenue is proportional to the total number of customers. Can we determine the total revenue for an arbitrary value of T, the number of tables we have? Note that an arriving customer will leave if all tables are occupied.

Problems such as this one can sometimes be solved analytically, i.e. we can write the expected revenue as a reasonably simple, easily evaluable function of the different parameters. But this is often not possible if the probability model is complex. For example, in the above description, we implied that the eating time probability distribution $e(t)$ is a function only of t. But if there are many people in the restaurant, the service might will be slower and each customer will occupy the table for longer periods. Thus, perhaps the distribution should be a function of the number of customers present as well. In this case, it will be much more difficult to write down an analytical solution. In such cases, a common

strategy is to simulate the system. By this, we mean the following. We pick random numbers from appropriate distributions to decide when customers arrive, how long they wait. Using this information, we compute how many tables are occupied at each instant, which customers need to be turned away because the restaurant is full, and so on. This can give us some understanding of "what might happen". If we run the simulation several times, each time with a different set of random numbers, we might be able to get some understanding of "what happens usually". Getting such understanding is the goal of performing simulations.

The restaurant simulation has a very different character from the cosmological simulation of Chapter 19. We will see that the restaurant simulation is an example of a *Discrete Event Simulation*. We will develop some machinery to perform discrete event simulations using which we will perform the restaurant simulation. We will also consider a variation, which we will call the *coffee-shop simulation*. The last topic in this chapter is the *shortest path problem* for graphs. We can get a fast algorithm for this abstract problem by viewing it as a simulation of a certain natural system. In Chapter 28, we develop a simulation of an airport, which also uses the machinery developed in this chapter.

27.1 DISCRETE-EVENT SIMULATION OVERVIEW

In a cosmological system, each star attracts, and thereby affects, every other star at every time instant. In contrast, in many other real-life systems the entities interact with each other relatively infrequently. For example, in a restaurant, a customer has interactions such as getting a table, ordering food, being served, paying the bill and leaving. These interactions are typically separated by long periods during which the customer is typically neither disturbed, nor demands any attention. Thus, it is customary to call this latter kind of system a *discrete time system*, whereas the former kind is called a *continuous time system*.

27.1.1 Discrete-time Systems

In general, a discrete-time system consists of a certain number of entities which might have states that can change over time. In addition, there is a list of enabled/impending events, i.e. events that are known will occur, each at some specified time in the future. When an event actually occurs, it may cause (a) the state of some of the entities to change, and (b) more events to be created. The new state depends upon the event that happened as well as the old state. Likewise, the new events that are created also depend upon the event that happened and the old state, i.e. the state of the entities just before the event happened. It is to be further noted that the state of a discrete system can change or new events created only during the occurrence of some event.

Many real-life systems can be considered discrete-time systems. For example, consider our restaurant system as described earlier. We have a very simple model in which customers approach and they are seated if there is a table available, and then leave after a certain randomly chosen time duration. If there is no table available, then we may consider that they do not even enter the restaurant. In this case, it is natural to think of the tables in the restaurant as the entities, and these can either be in the occupied or not occupied states. We may consider two kinds of events: customer *approach*, and customer *Exit*. Here is what happens during the occurrence of these events:

Approach: If all tables are occupied, then nothing happens. If there is an unoccupied table, then the number of occupied table increases by one, and an Exit event is created for occurrence at the current time + E, where E denotes the duration for which the customer eats.

Exit: The number of occupied tables reduces by one. No event is created for any future time.

S : State of the entities in the system. Initialized to the state at the current time.

L : List of events. Initialized to contain the events that are known will happen after the current time.

1. Let $e \in L$ be the event with the smallest time of occurrence.
2. Remove e from L.
3. Process e. This will cause change to S. New events may also be added to L.
4. Repeat from step 1 if L is not empty.

Fig. 27.1 *Discrete-event simulation algorithm*

As you can see, this description fits our notion of a discrete-time system. What happens during an events does depend upon the state of the world at that time. Further, the state of the system changes only because of events.

We will further note that it is customary to consider the *occurrence* of an event in discrete time systems to be instantaneous. This is not to say that actions such as eating are meant to be instantaneous. Indeed, as you saw above, eating starts with the approach event and finishes in the exit event.

We will see other examples shortly.

27.1.2 Evolution of a Discrete Time System

A key point to note is that if we know the state of a discrete time system at a given time and all the impending events, i.e. events that were created earlier but which havent yet occurred, we can precisely determine how the system evolves.

To see this, note that the current state of the system must persist until some event occurs. Thus, in particular, the current state must remain unchanged till the earliest of the impending events occurs. But we know the state at the time of this earliest event—it is simply the current state—and so we can precisely determine the effect of the earliest impending event. In other words, we will know how it will change the state and what new events will get created. But then we can keep going in the same manner, i.e. consider the next earliest event and determine its effect. The algorithm for this is abstractly given in Figure 27.1.

27.1.3 Implementing a Discrete-time System

The important question is, how should we represent the system state and events on a computer? Usually, the state of a system has a natural representation. For example, in our restaurant problem, we only need a single variable which keeps track of the number of occupied tables. How to represent events is more interesting.

When we say an event occurs, we typically have in mind a certain set of actions that takes place. Thus, it is natural to associate an event with a function which performs the required actions. The function must have access to the state of the system, and must be called only when the associated event becomes the earliest pending event, i.e. after earlier events have been processed. Thus, it turns out to be natural to represent each event by a lambda expression (Section 12.2), in particular a function that takes no arguments and returns no results. When it is time for the event to occur, we simply call the lambda expression!

As discussed in Section 12.2, the type of a lambda expression denoting a function that takes no arguments and returns no value is `std::function<void()>`. It will be convenient to assign a name to this type.

```
typedef std::function<void()> Event;
```

The mechanism of variable capture in lambda expressions will enable the function to get access to the system state, as we will see.

Second, for convenience, we will make the event list L of Figure 27.1 hold pairs of the form (e, t) where e is the event and t the time at which it is to happen. The (e, t) pairs that we need will be represented using the `pair` class in STL (found in the header file `<util>`). We use the type `double` to represent time, so we need a pair of `Event` and `double`. To simplify the subsequent discussion it is convenient to define

```
typedef pair<Event,double> ETpair;
```

Remember that the event list, consisting of pairs (e, t), is accessed in a very special manner. As seen in Figure 27.1, we do not remove arbitrary pairs from the list but only those with smallest t, though we may insert pairs in any order.

The `priority_queue` template class in the standard library (Section 22.6) supports nearly this mode of operation. It has operations to insert elements and also return the "largest" element in the queue, where we can define what is largest. Here is the prototype for `priority_queue`.

```
template<class T,                                    // argument 1
         class C = vector<T>,                        // argument 2
         class cmp = less<typename C::value_type> // argument 3
        >  priority_queue;
```

A prototype of a template is similar to a function prototype: both give the arguments and their types, and possible default values. In this case, the `priority_queue` template takes 3 arguments. The first argument gives the type of the elements that will be put in the queue, in our case this is `ETpair`. The second argument `C` specifies the container class using which the elements in a priority queue will be stored. The default option is to use a `vector`, which we do not wish to change. The third argument `cmp` is used to decide what to return. It defaults to `less`, which is simply the operator `<`. Thus of all the elements stored in it a priority queue returns that element `x` such that there is no `y` such that `x < y`. Thus a largest element is returned. To get a smallest instead, we must supply a function object equivalent of the `>` operator. We will see how to do this shortly.

The class `sim` shown in Figure 27.2 is the main simulation class. The member `pq` in it holds the event list implemented as a priority queue. Note that to use priority queues, it is necessary to include the header file `<queue>`, as shown at the top of the figure. The class `compareETpair` is defined to have a function call operator defined to implement the `>` operator on `ETpair` objects. So this class is supplied as the third element to the template defining `pq`. In addition to the event list, the class `sim` also has a member `time` which is meant to hold the time till which the system has been simulated. We start off by initializing time to 0.

This class `sim` is a little unusual. It is not meant to be instantiated! The members `time` and `pq` are defined as `static` elements. Thus, there will be only one copy of `pq` and `time`, but this is exactly what we want. The functionality is provided through four static member functions. Note that static member functions can be accessed outside the class definition as `class-name::function-name`.

```
#include <queue>            // needed for using priority_queue
class sim{
  typedef std::function<void()> Event;
  typedef pair<Event,double> ETpair;
  struct compareETpair{
    bool operator()(ETpair p, ETpair q){return p.second > q.second;}
  };
  static double time;  // time to which the system has been simulated
  static priority_queue< ETpair, vector<ETpair>, compareETpair> pq;

 public:
  static void post(double latency, Event e){
    pq.push(make_pair(e, time + latency));
  }

  static double getTime(){return time;}

  static void processAll(double tmax=1000){
    while(!pq.empty() && time < tmax){
      ETpair ETp = pq.top();   // ETp becomes (e,t)
      time = ETp.second;       // time becomes t
      pq.pop();
      ETp.first();             // call the event e
    }
  }

  static ostream & log(){
    cout << time << ") ";
    return cout;
  }
};

// Initialization of the static elements.

double sim::time = 0;
priority_queue< sim::ETpair, vector<sim::ETpair>,
sim::compareETpair> sim::pq;
```

Fig. 27.2 *The main simulation class*

The member function `post` allows an event to be posted, i.e. created and inserted into the priority queue. The first argument `latency` is the time duration after which the event is to happen, from the current time of the system. The second argument `e` is the event itself. The function thus inserts the pair `(e, time + latency)` into the priority queue.

The member function `getTime` merely returns the current time, i.e. the time till which the system has been simulated.

The member function `processAll` processes posted events till nothing is left to process. For this, it repeatedly picks the pair (e, t) in the priority queue with the smallest time t and calls the event `e`. Note that when we pick the element with the smallest time t, we know that the system time can be advanced to t. Thus the member `time` is updated to t. Note that the smallest element in the queue can be removed by using the `pop` method, or we can just examine it using the `top` method.

The last method, `log`, is for reporting convenience. It is used to print messages to the screen, but each message is prefaced by the current time. Note that a reference to the console output, `cout` is returned, so that the rest of the message can be appended using the « operator.

27.1.4 Simple Examples of Use

In general, to use any class such as `sim`, you must compile it along with your program, or include a suitable header file, etc. However, to simplify matters for you, we have included `sim` in `simplecpp` itself. Thus, you don't need to do anything other than include `simplecpp`.

We begin with some simple simulation examples. Given below is a program that just creates 2 events, say A,B and then asks the simulator to process them.

```
int main(){
  sim::post(15, [](){sim::log() << "Event A.\n";}); // event A
  sim::post(5, [](){sim::log() << "Event B.\n";}); // event B
  sim::processAll();                          // process all events.
}
```

The action associated with each event is simple: a message is printed, with the current simulation time.

When you execute, the first two statements will cause events A, B to be posted to occur 15 and 5 time units respectively from the current time. The simulation starts with time 0, so these events will occur at time 15 and 5 respectively. The third statement in the program will cause the system to be evolved, i.e. all the events in the queue will be processed, in the order of the time at which they are meant to occur. Thus event B, scheduled at time 5, will get processed, first, i.e. its lambda expression will be called. This will cause the message "Event B." to be printed, prefixed by the simulation time, because of the call to `sim::log`. After that event A will be processed. Thus, you would get the following output.

```
5) Event B.
15) Event A.
```

The point to be noted is that in the program the creation of event A happens before the creation of event B. However, the time of A, 15, is larger than the time of B, 5. Hence B happens before A during the execution.

As we discussed in Section 27.1.1, events in a discrete time system will, in general, cause the state of some of the entities to change, and also cause the creation (posting) of new events. We will now consider an example in which events do both of these. This example will be contain the core idiom used in all future simulations.

In the program below, we have 4 events, A, B, C, D. We also have a variable `count`. We will see that the events will access this variable ("simulation state"), and in fact event D will be posted during the occurrence of event C.

```
int main(){
 int count=0;
 sim::post(15, [&](){sim::log() << "A. count: "<< count++ <<endl;});
                                    //A
 sim::post(5, [&](){sim::log() << "B. count: "<< count++ <<endl;});
                                    //B
 sim::post(10, [&](){               //C
  sim::log() << "C. count "<< count++ <<endl;
  sim::post(100, [&](){sim::log() << "D. count: " <<   //D
                            count++ << endl;});
  });
  sim::processAll();
}
```

The program begins by setting `count` to 0, and then posting the events A, B, C for time 15, 5, 10 respectively. Then `sim::processAll` causes the posted events to be processed.

The event posted for the earliest time is event B, and hence that gets processed first. This will cause the time of occurrence of B to be printed and also the value of `count`. Note that `count` has been captured into the event by reference, and hence the value at the time of occurrence of the event will be printed. Thus, we will see the following output.

```
5) B. count: 0
```

Note that while printing, `count` is also incremented. Since `count` was captured by reference, the increment will affect the variable `count` as defined in the first line of the program. Thus this will become 1.

The event posted for the next earliest time, 10, is event C, which is then processed. The processing starts off by printing the message and the value of `count`. Notice that even in event C the variable `count` has been captured by reference. Thus the following message will be printed

```
10) C. count: 1
```

and `count` will be incremented. But the processing of event C does not stop after printing the message. After printing, the call `sim::post` is executed, which causes event D to be created, as a part of the occurrence of the event C. Event D is to occur at 100 steps after the current time, i.e. 10. Thus, event D gets posted for time 110. Thus, when the processing for C finishes, there are two events, A, D in the queue, and the value of `count` is 2.

Again, the earliest of the events is processed, i.e. event A. This will print a message

```
15) A. count: 2
```

and will increment `count` to 3.

After that event D will be processed. This will print a message:

```
110) D. count: 3
```

and cause count to increment to 4. Note the time in the message `sim::log` will indeed print 110, the time at which we expect the event to occur.

27.2 THE RESTAURANT SIMULATION

The restaurant operation depends upon the following parameters: the number `nTables` of tables, the time duration `duration` in minutes for which the restaurant remains open, the probability `arrivalP` that a customer arrives in the next minute, and the minimum and maximum eating times `eatMin, eatMax` for the customers. The restaurant state consists of the number of tables that are occupied, `nOccupied`.

At each minute that the restaurant is open, a customer can arrive with the specified probability. We model each arrival as an event. On arrival, if there are no unoccupied tables the customer leaves. If there are unoccupied tables, then the customer occupies one and sits down to eat. Then an eating duration is chosen uniformly randomly between `eatMin` and `eatMax`, and the customer must exit the restaurant after that duration. We model the exit also as an event. Thus, the arrival event for the customer must perform the above actions, including the creation of the exit event. All this goes into the lambda expression for the arrival event. On exit, it is only necessary to decrement the number of occupied tables. Thus, the decrementation code must be placed in the lambda expression of the exit event.

```
int main(){
  const int nTables=5;         // number of tables
  const int duration=180;       // minutes open
  const double arrivalP=0.1, eatMin=21, eatMax=40;
  int nOccupied = 0 ;           // number of tables occupied
  int id = 0;

  for(int t=0; t<duration; t++){
    if(randuv(0,1) <= arrivalP){// with probability arrivalP
      id++;
      sim::post(t, [=,&nOccupied](){// arrival event
        if(nOccupied >= nTables)   // if no table available
          sim::log()<< " Customer " << id << " disappointed.\n";
        else{                       // if a table is available
          ++nOccupied;
          int eatTime = randuv(eatMin, eatMax);
          sim::log()<<"  Customer " << id << " will eat for  "
                    <<eatTime<<"\n";
          sim::post(eatTime, [=,&nOccupied](){// exit event, posted
                                              // during arrival event
            sim::log()<< " Customer " << id << " finishes.\n";
            --nOccupied;
          });                       // exit event end
        }
      });                                   // arrival event end
    }
  }
  sim::processAll();
}
```

27.3 RESOURCES

Many real-life systems contain resources that are scarce. For example, machine might be needed to process several jobs of which it might be capable of processing only one at any instant. In such cases, the jobs must wait to gain exclusive access to the machine.

As an example, consider a roadside coffee shop manned by a single server. Suppose the shop serves beverages and food, all of which require some effort and time from the server. If a customer arrives while the server is busy with a previous customer, then the new customer must wait. So a line of waiting customers might form at popular coffee shops. Given the probability of customer arrival and the probability distribution of the service time, can we predict how much business the shop gets and also how long the line becomes?

We will develop a `Resource` class which will make it easy to write such simulations.

27.3.1 A `Resource` Class

So far, we have said that events can be created and posted to occur at a certain time in the future. We will extend this notion and allow an event to be posted to occur when a certain resource becomes available. If the resource is immediately available, then the event will occur immediately. Otherwise, it will be put in a queue associated with the resource. When the resource becomes available, because some other event releases it, the waiting event will be taken off the queue and posted for immediate execution.

These ideas are implemented in a `Resource` class.

```
class Resource{
  typedef std::function<void()> Event;
  queue<Event> q;
  bool inUse;
public:
  Resource(){inUse = false;}
  int size(){ return q.size(); }
  bool reserve(){
    if(inUse) return false;
    else{ inUse = true; return true;}
  }
  void acquire(Event e){
    if (!inUse){
      inUse = true;
      sim::post(0,e);
    }
    else
      q.push(e);
  }
  void release(){
    if(!q.empty()){
      Event e = q.front();
      q.pop();
```

```
      sim::post(0,e);
    }
    else inUse = false;
  }
};
```

We can create instances of the `Resource` class to model resources which must be used exclusively. The data member `inUse` in `Resource` when `true` denotes that the resource (represented by the instance) is deemed to be in use, and when `false` denotes that the resource is available. In addition, the resource class has a data member `q` which is a `queue` from the standard template library (can be used by including header `<queue>`). You can add elements to (the end of) a queue by calling the member function `push`, examine the element at the front of a queue by calling the member function `front`, and remove the element at the front by calling the member function `pop`.

The member function `acquire` can be used to get exclusive access to the resource. It takes as argument the event `e` that needs exclusive access, i.e. the lambda expression which will be executed when the resource becomes available. If the resource is immediately available, the event is posted for immediate execution by calling `sim::post(0,e)`. That the resource has been acquired is represented by setting `inUse = true`. If the resource is not available, then the event is put in the queue associated with the resource.

The member function `release` can be used to release a resource and make it available to other events. If an event `e` is waiting in the queue (front) of a resource that is being released, then `e` is posted for immediate execution by calling `sim::post(0,e)`. Note that in this case we do not need to change `inUse`: it was `true` before and must remain `true` because we assigned the resource to the waiting event. If there was no event waiting in the queue, then we must set `inUse = false`, so that a future `acquire` request can succeed immediately.

Finally, there is a member function `reserve` which marks the resource as being in use and returns `true` if and only if the resource was not in use earlier. We will see a use of this in Section 28.5.

27.3.2 Simple Example

The `Resource` class is also a part of Simplecpp, and so is immediately available for use. Here is a simple example.

```
int main(){
  Resource r;

  sim::post(15, [&](){
    r.acquire([](){sim::log() << "Got it! \n";});
  });

  r.acquire([&](){
    sim::post(20, [&](){ r.release();  });
  });
  sim::processAll();
}
```

The first statement posts an event for time 10, in which the resource `r` is sought to be acquired. Upon acquisition a message will be printed giving the time at which the acquisiton succeeds. The second statement seeks to acquire `r` immediately (at time 0), and releases it 20 steps after acquisition.

As you can see, the acquisition in the second statement will succeed, since at time 0, the resource is available. Thus, the resource will get released 20 steps afterwards, i.e. at time 20. Thus the acquisition in the first statement will succeed at time 20. Thus, that is when the message "Got it!" will be printed. Thus, the output of the program will be:

```
20) Got it!
```

Note that `r` needs to be captured in the second statement. Since `r` is shared between the statements, it is captured by reference.

27.3.3 The Coffee-shop Simulation

Using the `Resource` class, the simulation is easily written. The main program for simulating a 180-minute duration is as follows.

```
int main(){
  const int duration=180;                    // minutes open
  double arrivalP = 0.1, eatMin=3, eatMax=9;
  int id = 0;
  Resource server;

  for(int t=0; t<duration; t++){
    if(randuv(0,1) <= arrivalP){      // with probability arrivalP
      id++;
      int serviceT = randuv(eatMin, eatMax);
      sim::post(t, [=,&server](){
        sim::log() <<" "<< id << " arrives, service time "
                   << serviceT <<endl;
        server.acquire([=,&server](){
          sim::log() << " Customer: " << id << " being served.\n";
          sim::post(serviceT, [=,&server](){
            sim::log() << " Customer: " << id << " finishes.\n";
            server.release();
          });
        });
      });
    }
  }
  sim::processAll();
}
```

The general outline is similar to the restaurant simulation. We need a `Resource` to model the server, which is called `server` in the code. Then for each minute, we simulate customer arrival, with the arrival probability `arrivalP`. If a customer is deemed to arrive, then we generate a random service time for the customer. Then we must perform the following actions.

1. Print out a message saying that a customer arrives, along with the service time.
2. Request exclusive access to the server.
3. On getting exclusive access, we hold the server for the service time.
4. After the service time elapses, we release the server.

As you can see, the code above executes exactly these steps. However, the steps appear nested. This is because the code for an event is required to be placed in the argument list of the preceding event.

We note some important points about variable capture. First, `server` is shared across iterations, and hence must be captured by reference. However, we want the value of the variable `id` from the time of when the customer arrival is posted. Hence, `id` must be captured by value. Finally, `serviceT` is a local variable inside the loop, so it must be captured by value. Hence, we have asked that all variables be captured by value except for `server` which must be captured by reference.

27.4 SINGLE-SOURCE SHORTEST PATH

In this section, we consider the *single-source shortest-path* problem. Suppose we are given information about which cities are directly connected by road, and the length of all such roads. We want to travel from a given *origin* city to a *destination* city by road, passing whatever cities along the way, so that the total distance covered is as small as possible. The problem is difficult because there can be several paths from the origin to the destination, depending upon which cities you choose to pass through along the way. Of all such possible paths, we want the shortest. We will focus on the problem of finding the length of the shortest path, the path itself can be identified with a little additional book-keeping, which is left for the exercises. We discuss a classic algorithm, attributed to Edsgar Dijkstra. The algorithm actually finds the distance (length of the shortest path) from the origin to all other cities, because that is convenient.

Dijkstra's algorithm can be viewed as a computer analogue of the following physical experiment you could undertake to find the distances. For the experiment, we need many cyclists who can ride at some constant speed, say 1 km/minute. Specifically, we need to have as many cyclists in each city as there are roads leading out of it. If we do have such cyclists, here is how they could cooperatively find the length of the shortest paths.

To start with, all the cyclists assemble in their respective cities. Each cyclist is assigned one road leading out of the city, and the job of the cyclist will be to travel on just that one road when asked to. After the cyclist is flagged off somehow, she starts pedalling and reaches the city at the other end of the road. Here she must check if she is the first cyclist to arrive. If she is not the first, i.e. someone arrived earlier, then she does nothing and stops. If she is indeed the first to arrive, then she flags off all the waiting cyclists to start pedalling. After that her job is over.

Here is how the experiment starts off. At time 0, a fictitious cyclist arrives into the origin city, and flags off the cyclists in that city. They then flag off other cyclists as described above. The experiment ends when all cyclists have finished their journey.

As an example, suppose our graph is the map of Figure 27.3, and we want the distances from Nashik. So at time 0, a fictitious cyclist arrives into Nashik and flags off the cyclists there. So cyclists start pedalling from Nashik to respectively Nagpur, Mumbai and Pune. The cyclist from Nashik arrives at time 200 into Mumbai, where we are measuring time in minutes from the start. She is the first one to arrive there, so she flags off the 3 Mumbai cyclists who then start travelling towards Kolhapur, Pune, and Nashik respectively. Of these 3 the cyclist heading to Pune would reach 160 minutes later, at time

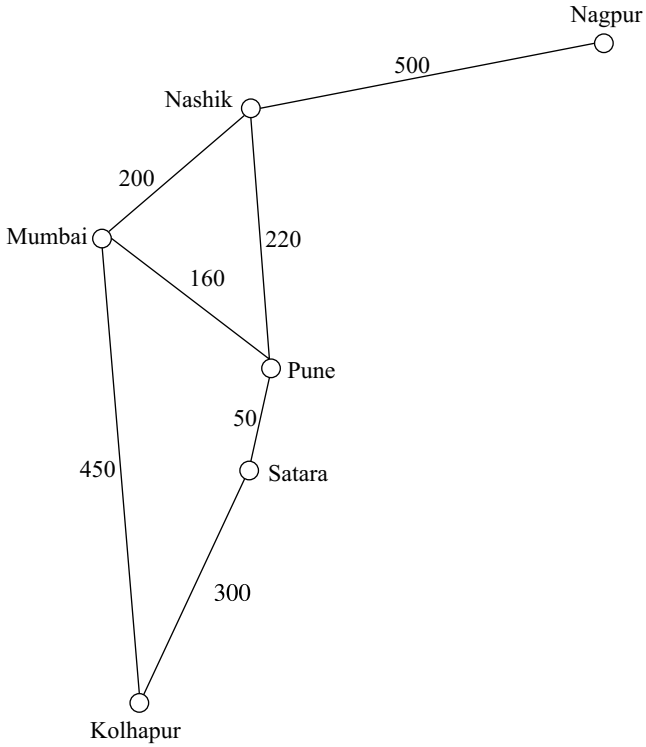


Fig. 27.3 *Schematic map*

360. However, when she reaches Pune, she would have found that the cyclist from Nashik has already arrived at time 220. So the cyclist arriving from Mumbai into Pune would need to do nothing. In this manner the process continues.

We will show that (a) the length of the shortest path from the source to any city is simply the time in minutes when the earliest cyclist arrives in that city! (b) We can use discrete event simulation to simulate this system.

We explain (a) first. Let S denote the source city, and C be any city. Let t be the time at which the first cyclist arrives into C. We argue that there must be a path from S to C of precisely this length. To see this, consider the cyclist that arrives into C. We follow this cyclist backward in time to the city from which she started. There, she was flagged off by some other cyclist, whom we follow back in time, and so on. Eventually, we must reach the city S, at time 0. In this process, note that we are not only going back in time but also continuously travelling back, at 1 km/minute. Thus, we must have covered, backwards, exactly the same distance as the time taken. Thus, we have proved that there exists a path from S to C of length equal to the time at which the first cyclist arrives in C. We now prove that it is the shortest.

Consider a shortest path P from S to C, the cities on it being $c_0, c_1, \ldots, c_k$ in order, with $c_0 = S$, and $c_k = C$. Let d_i be the distance from c_0 to c_i along the path. We will prove that the first cyclist (fictitious or real) to arrive at c_i does so no later than d_i, for all i. Clearly, this is true for $i = 0$: indeed a cyclist arrives at c_0 at $0 = d_0$. So assume by induction that a cyclist arrives at c_i at time d_i or before. Thus, a cyclist must have left c_i at time d_i or before for city c_{i+1}. But this cyclist travels at 1 km/minute,

and hence covers the distance $d_{i+1} - d_i$ also in time $d_{i+1} - d_i$. Hence, she will arrive at c_{i+1} at time at most $d_i + d_{i+1} - d_i = d_{i+1}$. Thus, the induction is complete. Thus, we know that some cyclist must arrive at $c_k = C$ at time at most the length of the shortest path P. But we proved earlier that the time of arrival must equal the length of some path. Hence, it follows that the first cyclist arrives at time exactly equal to the length of the shortest path.

```
struct City;       // forward declaration, not definition.
struct Road{
  City*  toPtr;    // Where the road leads to
  double length;
  Road(City* ptr, double d){toPtr = ptr; length = d;}
};

struct City{
  vector<Road> roads;
  double arrivalT;               // arrival time of first cyclist
  City(){arrivalT = HUGE_VAL;}   // not arrived yet.
  void arrive(){
    if(arrivalT > sim::getTime()){
      arrivalT = sim::getTime();
      for(unsigned int i=0; i<roads.size(); i++){
        sim::post(roads[i].length, [this,i]()
          {roads[i].toPtr->arrive();});
      }
    }
  }
};

struct RoadNetwork{
  vector<City> cities;
  RoadNetwork(char* infilename) {
    ifstream infile(infilename);
    int n;
    infile >> n;
    cities.resize(n);
    double dist;
    int end1, end2;
    while(infile >> end1){
      infile >> end2 >> dist;
      cities[end1].roads.push_back(Road(&cities[end2],dist));
      cities[end2].roads.push_back(Road(&cities[end1],dist));
    }
  }
};
```

Fig. 27.4 *Graph representation*

We next show that our algorithm can be programmed as a discrete event simulation.

27.4.1 Dijkstra's Algorithm as a Simulation

The first question, of course, is how to represent our network of roads. The network is a graph, in which the cities are the vertices and the roads the edges. So we use the representation as given in Section 23.2.3. This is shown in Figure 27.4.

The entire road network is held inside the class `RoadNetwork`. It contains a vector, `cities`, the `i`th element of which is an object of class `City` containing information about the `i`th city. Thus, we must assign a number to each city in our map. For our map of Figure 27.3, we assign the numbers 0 to 5 to the cities Kolhapur, Mumbai, Pune, Nashik, Nagpur, Satara respectively. The member `arrivalT` in each `City` object is meant for storing the time at which the first cyclist arrives into that city. Each `City` object also contains a vector `roads` which stores information about the roads leaving that city. Suppose `G` is a `RoadNetwork`. Then `G.cities[i].roads[j]` is an object of type `Road` which stores information about the `j`th road leaving city `i`. Specifically, the object stores the following: (a) a pointer `toPtr` to the city that this road leads to, (b) a double `length` giving the length of this road.

The constructor for the class `RoadNetwork` reads in the road network from the file whose name is given as an argument. Figure 27.5 shows a sample input file. This file represents the road network of Figure 27.3. The first number in the file gives the number of cities. On reading this the constructor resizes the vector `cities` to this number. This will cause elements of the vector `cities` to be created. Thus a `City` object is created for each city in the network. Note the constructor for `City`: it sets `arrivalT` to `HUGE_VAL`, which represents ∞. We use this to denote that as of now, no cyclist has arrived into the city. Next, the constructor of `RoadNetwork` reads information about the roads in the graph. This consists of triples `c1, c2, dist`, where `c1`, `c2` give the cities at the two ends of a road, and `dist` gives the length of the road. We must store the information about this road in the structure `cities[c1]` which stores information related to city `c1`, as well as in `cities[c2]` which stores information related to city `c2`. That is done in the two statements in the loop. When the loop finishes, the road network will have been constructed.

The main program creates the graph, and starts off the simulation of the movement of the cyclists, as shown below.

```
int main(int argc, char** argv){
  RoadNetwork G(argv[1]); // create road network from file
  int origin;          // city from which distances are needed
  stringstream(argv[2]) >> origin;

  // cause the fictitious cyclist to arrive.
  sim::post(0, [&G,origin](){G.cities[origin].arrive();});
  sim::processAll();

  for(unsigned i=0; i<G.cities.size(); i++)
    cout << G.cities[i].arrivalT << " ";
                // arrivalD = distance from origin
  cout << endl;
}
```

File content	Explanation
`6`	Number of cities
`0 1 450`	Kolhapur Mumbai distance
`0 5 300`	Kolhapur Satara distance
`1 2 160`	Mumbai Pune distance
`1 3 200`	Mumbai Nashik distance
`2 3 220`	Pune Nashik distance
`3 4 500`	Nashik Nagpur distance
`5 2 50`	Satara Pune distance

Fig. 27.5 *Input file for graph of Figure 27.3*

The program uses command-line arguments. The first command-line argument `argv[1]` gives the name of the file which contains data to build the road network. We supply this file name to a constructor of the class `RoadNetwork` which builds the object `G` for us. The second command line argument, `argv[2]` is expected to be an integer, and it gives the index of the origin city. For this, we first convert the string `argv[2]` to a `stringstream` (Appendix E), and then read from it. Then we start off the simulation.

To start off the simulation, we must flag off the cyclists in the `origin` city. For this, we post an arrival event at time 0 for the origin city. As you will see below, this arrival causes cyclists to move and arrive into other cities and so on. After posting the arrival into the origing city, `main` merely waits for all events to be processed, i.e. for all cyclists to finish their journey. At the end the earliest time to reach each city `i` from the city `origin` can be found in `G.cities[i].arrivalT`. But that is also the distance, and so it is printed.

There is only one kind of event in the simulation: the arrival of a cyclist into a city. The actions to be taken during the event are placed in the member function `City::arrive`. To check if the arriving cyclist is the first cyclist to arrive into the city, we examine member `arrivalT` in `City`. If `arrivalT` is not `HUGE_VAL`, then it has changed since the city was created, i.e. the cyclist is not the first to arrive. In this case, we do nothing. On the other hand, if `arrivalT` is still `HUGE_VAL`, then this cyclist is the first to arrive, and the following actions must be performed.

1. Record the correct arrival time into `arrivalT`. Note that after this that it will no longer be `HUGE_VAL`.
2. Cyclists must be flagged off to leave the city on each outgoing road `i`. The cyclist will reach the corresponding city, pointed to by `roads[i].toPtr`, after covering the distance `roads[i].length`, i.e. after that much time. Hence, we post an arrival event for that city with that much latency.

As you can see, a single cyclist arriving into the origin city will indeed eventually cause all cyclists to start moving. Thus, the shortest distance will get computed.

27.5 CONCLUDING REMARKS

Discrete-event simulation is a powerful paradigm. Using the classes developed in this chapter, you should be able to easily build some modest size simulations. We will see such an example in the next chapter.

The basic ideas in discrete event simulation are as follows. Whatever system you wish to simulate must first be viewed as a collection of interacting entities. The interactions constitute the events. Each event can cause the state of an entity to change, or cause additional future events to be posted. We express each event as a lambda expression. When the event happens, the associated lambda expression executes, and in this execution we must change the state of the entities in the system or post additional events. Note that the code for doing all this could be placed textually inside the lambda expression, as was the case in the restaurant and coffee shop simulations. Or inside the lambda expressions, we can just place a call to a (member) function which causes the state changes or posting of additional events, as was the case in the shortest path algorithm simulation.

The coding style for posting events and acquiring resources is slightly tricky. Normally, when we wish to perform one action after another, we write the second action following the first. However, if event A causes event B which in turn causes event C, then we would write the lambda expression of C inside the lambda expression of B which in turn could be in the lamdba expression of A. Thus, although the events happen in succession, in the code they will appear nested. This needs some getting used to.

Note that lambda expressions make it convenient to express event posting and resource acquisition. You should make sure that you understand lambda expressions well, especially variable capture.

EXERCISES

1. Modify the restaurant simulation to report how many customers left disappointed, how long after the closing time did the customers stay around, the number of customers in the restaurant on the average.
2. Generalize the coffee shop problem so that there are several servers. This is also like adding a waiting room to the restaurant. You will need to modify `resource`. Generalize the class so that at most some k clients can be using the resource simultaneously. You may find it easier to do this if you do not keep track of which clients are using the resource, but just keep track of how many clients are using the resource.
3. Suppose every minute a customer enters a store with a probability p. Suppose that on the average each customer spends t minutes in the store. Then on the average, how many customers will you expect to see in the store? *Little's law* from queueing theory says that this number will be pt. Modify the coffee shop simulation and verify Little's law experimentally. The law requires that no customers are turned away, and that the average is taken over a long (really infinite) time. So run the simulation for relatively long durations. More code will be needed to make all the measurements.
4. Write a simulation of a restaurant in which customers can arrive in a group, rather than individually. Suppose a group can have up to 5 members, all sizes equally likely. Suppose further that tables in the restaurant can accommodate 4 customers, so if a party of 5 arrives, then two adjacent tables must be allocated. Thus, the party must wait if two adjacent free tables are not available. Write a simulation of such a restaurant. Assume that the tables are in a single line, so

tables $i, i+1$ are adjacent. You will have to decide on how a table will be allocated if several tables are free: this will affect how quickly you serve parties of 5 members.

5. Have an additional command-line argument which gives the index of a *destination* city, for the shortest path program. Modify the program so that it prints the shortest path from the source to the destination city, as a sequence of the numbers of the cities on the way. Basically, in each `City` you must store information about where the first cyclist arrived from. This will enable you to figure out how the shortest path arrives into a `City`, recursively.

6. Modify the shortest path algorithm to use city names instead of city numbers in the input file.

7. Build a simulator for a circuit built using logic gates. Consider the gates described in Exercise 16 of Chapter 6. You should allow the user to build the circuit on the graphics window. You should also allow a delay δ to be entered for each gate. A gate takes as input values 1 or 0, and produces output values according to its function. However, the output value is reliably available only after its delay. Specifically, suppose some input value changes at time t. Suppose this will cause the output value to change. Then the new correct value will appear at the output only at time $t + \delta$. During the period from t to $t + \delta$, the value at the output will be undefined. For this you should use the value `NAN` supported as a part of the header file `<cmath>`. The value `NAN` represents "undefined value", actually the name is an acronym for "Not A Number". This value behaves as you might expect: do any arithmetic with it and the result is `NAN`.

8. Paul Erdos was an extraordinary Hungarian mathematician. As a humorous tribute to him, an Erdos number is defined for every person. Erdos himself has Erdos number 0. The numbers for other individuals are defined recursively as fellows. The Erdos number for a person X is $n + 1$, if n is the minimum Erdos number amongst the numbers of the collaborators of X. Thus, all collaborators of Erdos get Erdos number 1. All their collaborators (who havent yet been given a number) get the number 2, and so on. Write a program which takes information about who has collaborated with whom, and prints the Erdos number of each. The input to your program will consist of N the number of persons, followed by pairs of integers between 0 and $N - 1$. Each pair of integers x, y indicates that x, y are collaborators. Assume that person numbered 0 represents Erdos.

CHAPTER 28

Simulation of an Airport

Suppose there are complaints about the efficiency of an airport in your city: say flights get delayed a lot. Is it possible to pinpoint the reason? Is it then possible to state the best cure to the problem: that you need to build an extra runway, or some extra gates, or perhaps just build a completely new, bigger airport? A simulation of the airport and how it handles aircraft traffic can very much help in making such decisions.

The simulation will take as input information about the runways and other facilities on the airport, and about the aircraft arriving into the airport from the rest of the world. It will then determine what happens to the aircraft as they move through the airport, what delays they face at different points. The sum of these delays is perhaps an indicator of the efficiency of the airport. To answer questions such as: how much will an extra gate (or runway or whatever) help, you simply build another simulation in which the extra gate is present, and calculate the average delay for the new configuration. In addition to textually describing what happens to each aircraft as it progresses through the airport, it is also desirable to show a graphical animation in which we can see the aircraft landing, taxiing or waiting at gates. An animation is possibly easier to grasp – perhaps seeing the aircraft as they move might directly reveal what the bottlenecks are.

The first step in building a simulation is to make a computer model of the relevant aspects of the system being simulated. When you make a computer model, or a mathematical model, of any entity, doubtless you have to throw away many details. A trivial example: the colour of the airport building is irrelevant as far as its ability to handle traffic, so that may be ignored in our simulation. On the other hand, the number of runways in the airport is of prime importance, and so cannot be ignored. Other factors that perhaps cannot be ignored include the number of gates at which aircraft can park to take in and discharge passengers, the layout of the taxiways that connect the runways and the terminals. Other factors that are perhaps less important are the placement of auxiliary services (e.g. aircraft hangars) and traffic associated with these services and how it might interfere with aircraft movements. In general, the more details you incorporate into your model, the more accurate it is likely to be. However, models with relatively few details might also be useful, if the details are chosen carefully.

In this chapter, we will build a simulation of a simple airport. The simulation is very simplistic, but it does address several key problems that arise in such simulations.

28.0.1 Chapter Outline

We will begin by describing the specification, i.e. what we plan to simulate. We discuss the airport configuration and the (simplified) rules under which the airport will be required to operate. Then we present an implementation. An important problem in simulating complex systems such as an airport is *deadlock*. We discuss how deadlocks can be dealt with in real life and in programs.

The code for the simulation is given with Simplecpp. Figure 28.1 is a snapshot from its execution.

28.1 THE SIMULATION MODEL

It is possible to write a simulator which simulates airports with arbitrary number of runways, taxiways, gates and so on. However, for simplicity, we will consider the specific airport configuration shown in Figure 28.1.

In the figure, the two crossing lines at the top are two runways. The other lines are taxiways. The long horizontal line near the bottom is the main taxiway, and the nearly vertical segments on the sides we will refer to as the left and right taxiways respectively. There are branches going off the main taxiway to the gates. We have not shown the gates, but they are supposed to be present at the end of these short branches. So in this airport there are meant to be 10 gates, which we will number 0 through 9, right to left. The small triangles are meant to represent aircraft. As you can see, there are three aircraft waiting, at gates 0, 1, and 3, and three others on the runway and taxiways. If you ignore the branch taxiways, the runways and the other taxiways constitute a single long path, starting in the top left corner, running clockwise over itself to end in the top right corner. We will call this the *main path*. Indeed, for simplicity, we will require that the main path be used in the clockwise direction. Thus, the runway starting at the top left is the landing runway and the runway ending at the top right is the takeoff runway. The branch taxiways going to the gates are expected to be used in both directions.

Our configuration is rather simplistic, except for the intersecting runways. Intersecting runways are not rare, by the way—in fact the Mumbai airport has intersecting runways, which is our inspiration for including them. But, of course, both the runways in Mumbai can be used for takeoffs as well as landings, and the taxiways and gate placements are more elaborate.

28.1.1 Overall Functioning

At a high level, the operation of an airport can be described as follows. Each aircraft lands and taxies to a gate. The aircraft then waits at the gate for a certain *service* time. After that the aircraft taxies to the

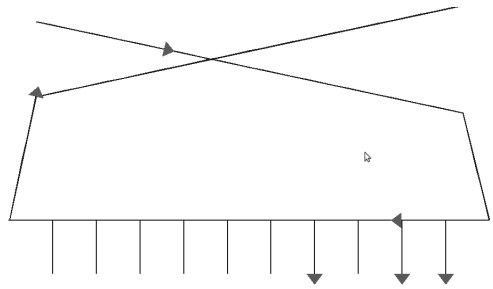


Fig. 28.1 *Airport layout with planes*

runway and takes off. This entire process has to be controlled by the airport authorities so as to ensure safety and efficiency.

28.1.2 Safe Operation and Half-runway Exclusion

The gist of the safety requirements is: aircraft movement should be planned so that at all times aircraft are well separated from each other. A certain minimum separation is required even as aircraft are taxiing. The separation between aircraft must be larger when they are travelling at high speeds, as will be the case when they are landing or taking off. The separation might depend upon the type/size of the aircraft. For simplicity, we will assume that there is just one type of aircraft and ignore this issue.

More formally, we will model the safety requirement as follows: we will break runways and taxiways into segments and require that there be at most one aircraft on each segment at any time. Thus, by choosing sufficiently long segments we can keep the aircraft well separated. Here is the division into segments that we will use. The two runways will be separate segments, and so will the the left and right taxiways. The main taxiway will be broken up into segments at the points where the branch taxiways leave from it. Since there are 10 gates, the main horizontal taxiway will be split into 11 segments. The branch taxiways will constitute separate segments by themselves.

We have an additional complication because our two runways overlap. The simplest way to ensure safe operation would be to say that only one of the runways can be used at any time. However, to make the problem more interesting and realistic, we will note that the intersection is in the initial portion of the runways, and so we will require that the initial halves of the runways should not be in use simultaneously. In other words, we will require that if the initial half of the take off runway contains an aircraft then there should be no aircraft in the initial half of the landing runway, and vice versa. We will call this *half-runway exclusion*.

28.1.3 Scheduling Strategy

The exact schedule according to which aircraft land and takeoff and even move around while on the airport is decided by the air traffic controllers at the airport. They must obey the safe operation rules and in addition resolve conflicting requests. For example, if two aircraft request permission to use the runway (either for take off or for landing) at the same time, then permission can be granted to only one. Such decision will have to be taken by the air traffic controllers, so as to acheive certain goals, e.g. to minimize the average delay, or some weighted average delay with the weights being the priorities of the different aircraft.

We will assume that a very simple *first come first served* scheduling strategy is being used by the air-traffic controllers. Basically, each aircraft requests permission from the traffic controller for each action it needs to perform, just as it becomes ready to perform the action. If several aircraft ask permissions to perform actions which require a common resource (say the runway), then permission is granted to the aircraft which asked earliest, and the other aircraft must wait. Of course many other strategies are possible. For example, we might decide to give higher priority to landings than takeoffs because it is easier for a plane to wait on ground than wait mid air![1] This is explored in an exercise.

[1] An aircraft must begin its descent much earlier than its landing time, and once the descent has begun, the landing cannot be postponed in normal circumstances. However, our first come first serve strategy may require a flight arrival to be delayed. This is a shortcoming of our simulation.

28.1.4 Gate Allocation

When an aircraft arrives it must be assigned a gate at which it is to wait. In general, each aircraft may have its preferred gates at which it would like to wait. For simplicity, we will assume that all aircraft can wait at all gates, and say the least numbered free gate should be allocated.

Gates can be allocated any time after the plane arrives into the airport. We will assume for definiteness that that gate allocation must be done just when the aircraft is about to turn into the main taxiway from the right taxiway.

28.1.5 Simulator Input and Output

The input to the simulator consists of two number for each incoming aircraft: the arrival time, and the service time, i.e. the amount of time the aircraft needs to wait at a gate. The simulation will need to have information about how long it takes for an aircraft to traverse the runway and taxiways, but we will consider this to be a part of the program.

The primary output from the simulator will be (a) an animation of the aircraft as they enter the airport, move to a gate, halt for the required time, and then take off and leave, and (b) a text record of the times at which these events happen. When designing an animation, we need to decide how frequently will we show the state of our airport. Do we show it every second, or every minute, or only when something interesting happens, e.g. an aircraft arrives or leaves or stops at its gate? For simplicity, we wil assume the state is to be shown after every unit time interval, whatever the unit time we define in the program.

In addition, we may require several derived outputs. Let us define the delay of an aircraft to be the additional time it spent over and above when it could have departed had the airport been completely empty. So we might be required to compute the average delay. Such analyses and extensions are left to the exercises.

28.2 IMPLEMENTATION OVERVIEW

We will build a discrete-event simulation using the `sim` class developed in Chapter 27. You will see that the `Resource` class developed there will also come in useful.

This simulation belongs to the following general paradigm. We have a system (airport) into which some client entities (aircraft) enter. The entities need to perform certain activities, after which they leave the system. The time required for the different activities may be specified (e.g. how long does it take to traverse a taxiway, how long should the aircraft wait at a gate). However, it may be necessary that the client entities get exclusive access to certain resources in order to perform the activities. In the case of the airport, each plane needs exclusive access to a gate. However, because of the need to keep safe distance, we have decided that there must be only one plane on any taxiway at any step. Thus, before moving onto a taxiway, a plane must get exclusive access to the taxiway, i.e. each taxiway must be treated as a resource. As you can guess, the `Resource` class will be useful for representing taxiways (and also gates).

Once the resources are identified, we can describe simulation from the point of view of each client. The initial event is the entry of the client into the system. After that the client tries to acquire the required resources and make progress through the system. If a certain resource is not available, an event will be queued up at the resource, and the activity of the client will get suspended. When the resource becomes available, the event will get processed, which will resume the activity of the client. Likewise, if a certain activity needs a certain large duration (e.g. an aircraft must wait at the gate for its

service time), then we can stop execution for now after posting an event which will wake us up after later. This is the general strategy followed in the implementation below.

28.2.1 Half-runway Exclusion

To implement half-runway exclusion we will use the following trick. Whenever a plane needs to land or take-off, we will require it to reserve a fictitious `halfRW` taxiway in addition to reserving the landing or take-off runways respectively. After a plane has landed and traversed half the runway, we release `halfRW`. Thus, another plane can to start taking off if needed. Same thing for a plane taking off—it will also release `halfRW` when it gets to the middle of the take-off runway.

28.2.2 Gate Representation and Allocation

We represent gate `G` implicitly using branch taxiway `G`. Indeed, when a plane has to wait at gate `G` it waits at the bottom end of branch taxiway `G`. Furthermore, when we want to allocate gate `G` to a plane, we merely reserve branch taxiway `G`. With this we can ensure that at anytime a gate is used only by one plane.

To allocate a gate we merely examine all the branch taxiways and determine if any is free, and if so reserve it. The plane then taxies to the end of that branch taxiway and waits. After waiting for the service time decided for the plane, the plane turns around and heads back to the main taxiway. Just as the plane is about to turn onto the main taxiway it releases the reservation for the branch taxiway. After this the other planes can use this gate.

28.3 MAIN PROGRAM AND DATA STRUCTURES

We represent aircraft using a class `plane`. Taxiways and runways are functionally equivalent, and are represented using a class `taxiway`.

The main program causes the creation of the taxiway and plane objects and the posting of arrival events for them. After that `sim::processAll` is called and the simulation unfolds starting from the arrival events. The details appear in the `plane` class (Section 28.5).

```
const int nGates = 10, nSegments = 6+3*nGates;

int main(int argc, char** argv){
  vector<taxiway*> taxiways(nSegments);  // including halfRW
  initCanvas("Airport Simulator",1000,1000);

  configure_taxiways_and_runways(taxiways);
  ifstream planeDataFile(argv[1]);
  post_plane_arrivals(taxiways, planeDataFile);
  sim::processAll();
  getClick();
}
```

The function `configure_taxiways_and_runways` sets up the vector `taxiways` that holds all required taxiways and runways. We discuss this function in the next section, which also discusses the `taxiway` class in detail.

The function post_plane_arrivals reads the arrival time and the service time for each plane from a file supplied as the first command-line argument and creates the plane objects. Note that planes need to move on the taxiways, and so the vector taxiway is supplied as an argument to the plane constructor. After each plane is created, the constructor posts an arrival event for the designated arrival time. The details of the plane class are discussed later.

```
void post_plane_arrivals(vector<taxiway*> &taxiways, ifstream
  &planeDataFile){
  int arrivalT, serviceT;
  int id = 1;                  // identifier for each plane
  while(planeDataFile >> arrivalT){
    planeDataFile >> serviceT;
    new plane(id++, arrivalT, serviceT, taxiways);
                               //this posts arrival
  }
}
```

28.4 THE taxiway CLASS

Instances of the taxiway class must serve two purposes: they must be visible on the screen as lines, and the planes must be able to reserve them. So it is natural to derive the taxiway class from the Line class and the Resource class of the preceding chapter.

```
class taxiway : public Line, public Resource{
public:
  int traversalT;         // time needed to traverse the taxiway
  double stepsize;        // distance covered per time step
  taxiway(float xa, float ya, float xb, float yb, int trT)
    : Line(xa,ya,xb,yb), traversalT(trT),
    stepsize(sqrt(pow(xa-xb,2)+pow(ya-yb,2))/traversalT) {}
};
```

The taxiway constructor first creates the Line representing the taxiway on the screen. Ideally we should distinguish the on-screen line from the real taxiway, and provide details about the real taxiway separately. For simplicity, we have assumed that the on-screen taxiway and the real taxiway will have same coordinates on the screen as well as the ground (say the units have been conveniently selected). In constructing a taxiway, we also provide the time required to traverse it in some hypothetical time units. Since we know the length of the taxiway, we calculate how much an aircraft moves forward each (hypothetical) step when on this taxiway—this information is needed to perform the animation.

Note that the Resource constructor is not explicitly called, so a call with no arguments will be inserted by the compiler. This will set the member inUse of the taxiway (derived from Resource, Section 27.3) to false, indicating that initially the taxiway is unreserved.

The two runways, the taxiways and the fictitious taxiway halfRW are represented by taxiway objects, organized into a vector taxiways. This vector is created by the function configure_taxiways_and_runways. Although we simulate an airport with 10 gates, it is be convenient to express the creation using a parameter nGates = 10. You will see that we

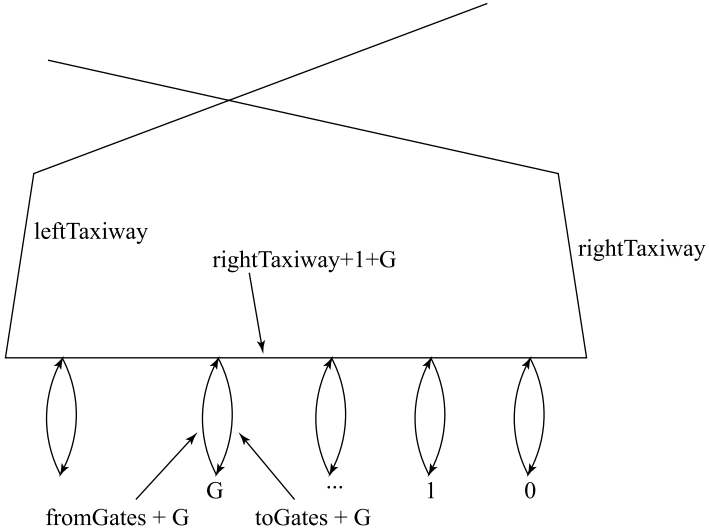


Fig. 28.2 *Gate and segment numbering*

need `3*nGates + 6` segment taxiways including `halfRW`. The initial `nGates+5` elements of `taxiways` respectively represent the landing runway, the right taxiway, the `nGates+1` segments of the main taxiway, the left taxiway, and the take-off runway (Figure 28.2). The `nGates` subsequent elements will represent the branch taxiways going toward the gates, and the next `nGates` elements will represent the branch taxiways coming back from the gates. Then we will have one more segment representing the fictitious taxiway `halfRW`.

The code below creates the `taxiway` elements along with their geometrical coordinates for display purposes. The names `RW1X1`, etc., are constants indicating the geometric coordinates of the appropriate taxiways, and the names `tRW`, etc., are constants indicating the time to traverse the appropriate taxiways.

```
void configure_taxiways_and_runways(vector<taxiway*> &taxiways){
  taxiways[0] = new taxiway(RW1X1,RW1Y1,RW1X2,RW1Y2,tRW);
                 // landing runway
  taxiways[1] = new taxiway(RW1X2,RW1Y2,TWX1,TWY1,tVT);
                // right taxiway

  float twXdisp = ((float)TWX2-TWX1)/(nGates+1);
  float twYdisp = ((float)TWY2-TWY1)/(nGates+1);

  for(int i=0; i<= nGates; ++i){
              // main taxiway: nGates+1 segments
    taxiways[2+i] = new taxiway(int(TWX1+i*twXdisp),
                                int(TWY1+i*twYdisp),
                                int(TWX1+(i+1)*twXdisp),
                                int(TWY1+(i+1)*twYdisp), tMT);
  }
```

```
  taxiways[3+nGates] = new taxiway(TWX2,TWY2,RW2X1,RW2Y1,tVT);
              // left taxiway
  taxiways[4+nGates] = new taxiway(RW2X1,RW2Y1,RW2X2,RW2Y2,tRW);
                // takeoff runway

  for(int i=0; i<nGates; ++i){       // branches to gates
    taxiways[5+nGates+i] = new taxiway(int(TWX1+(i+1)*twXdisp),
                  int(TWY1+(i+1)*twYdisp),
                  int(TWX1+(i+1)*twXdisp), TWYT, tBT);
  }
  for(int i=0; i< nGates; ++i){             // branches from gates
    taxiways[5+2*nGates+i] = new taxiway(int(TWX1+(i+1)*twXdisp),
                           TWYT,
                           int(TWX1+(i+1)*twXdisp),
                           int(TWY1+(i+1)*twYdisp), tBT);
  }
  taxiways[5+3*nGates] = new taxiway(0,0,0,0,0);  // halfRW
}
```

It will be convenient to define the following names for taxiway segment indices.

```
const int toGates = 5+nGates, fromGates = 5+2*nGates,
  halfRW = 5 + 3*nGates;
const int preLanding = -1, landing = 0, rightTaxiway = 1,
  leftTaxiway = toGates-2, takeOff = toGates-1;
```

The taxiway numbering vis-a-vis gate numbering is shown in Figure 28.2. We have shown the taxiway going to a gate as being distinct from the taxiway going back from the gate. This is only to stress that each physical taxiway connecting to the gate (Figure 28.1) is modelled as two logical taxiways, one in each direction.

28.5 THE `plane` CLASS

The aircraft are implemented using a `plane` class. An aircraft must appear on the screen as a part of the animation. So we inherit from the `Turtle` class. Indeed, our aircraft appear on the screen as turtles. We could have defined a more aircraft like visual appearance, but that is left for the exercises. To keep track of which taxiway segment the plane is on at any time instant we will have a data member `segment`. We will have another data member `timeToSegmentEnd`, in which we keep track of the number of steps we need to move forward in order to reach the end of the current segment. In addition, we need to note the gate allocated for the aircraft. For this, we use an integer data member `gate`. Here is the class.

```
class plane : public Turtle {
  int id;          // identifying number for plane
  int arrivalT;    // arrival time
  int serviceT;    // how long the aircraft waits at the gate
  int segment;     // index of taxiway segment the aircraft is on
```

```
  int timeToSegmentEnd;  // how far from the end of the segment
  int gate;        // id of allocated gate
  vector<taxiway*> &taxiways;
public:
  plane(int i, int at, int st, vector<taxiway*> &tw)
    : id(i), arrivalT(at), serviceT(st), taxiways(tw) {
    segment = preLanding; // currently before the landing runway.
    timeToSegmentEnd = 0;
    hide();
    penUp();
    gate = -10*nGates;   // indicates gate not allocated
    sim::post(arrivalT, [=](){prepareToEnterSegment();});
  }
  void prepareToEnterSegment();
  void land();
  void enter(int newsegment);
  void moveOnSegment();
  void requestGate();
  void turnToGate();
  void atGate();
  void backOnTaxiway();
  void prepareToTakeOff();
  void depart();
  void ordinarySegment();
};
```

The constructor initializes the data members. The first event associated with a plane is its arrival into the airport. This must happens at time `arrivalT`. Thus the constructor posts an event for this time. When time advances to `arrivalT`, this event will get processed and `prepareToEnterSegment` will get called. Note that in this call `this` has been captured (Section 17.5.4).

The other member functions implement the actions of the plane as it moves through the airport. It is convenient to think of the entire lifetime of a plane as repeated executions of the two main steps below.

1. Preparing to enter a taxiway segment. This involves acquiring different resources depending upon the segment being entered.
2. Movement along the segment.

The action of preparing to enter a taxiway segment is handled by the member function `prepareToEnterSegment`. The specific action depends upon what segment is being entered. For example, if `segment` has the value `preLanding` as it would have initially, the function `land` will be called.

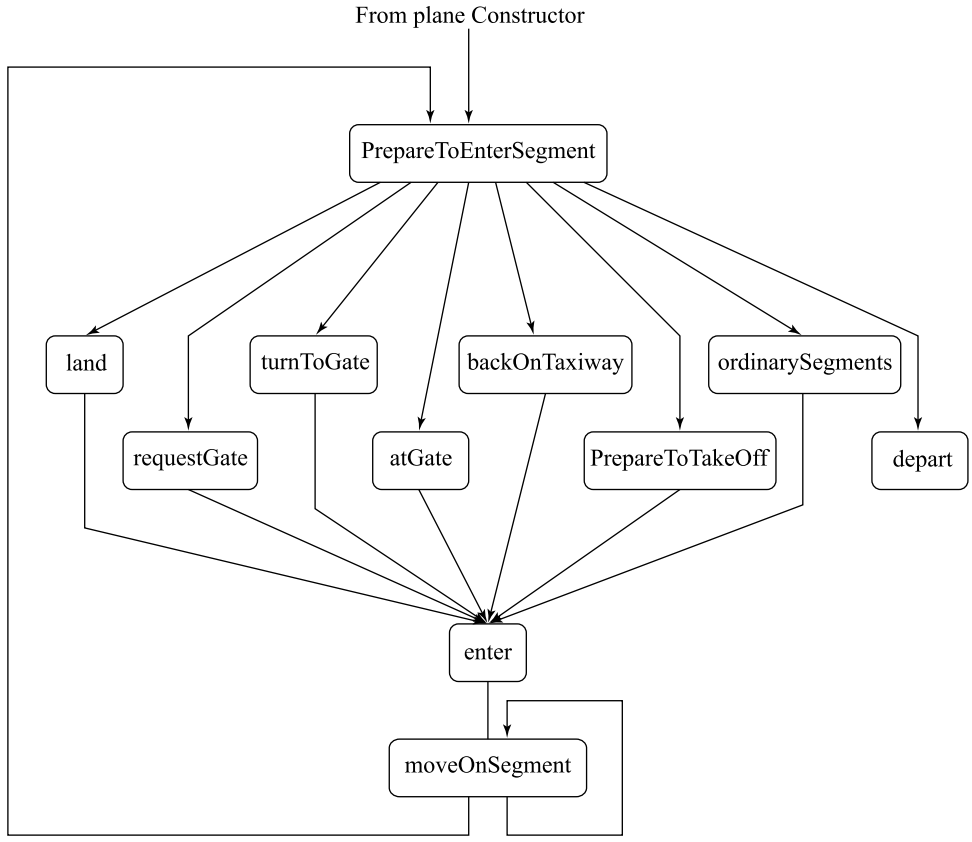


Fig. 28.3 *Plane-action block diagram*

```
void plane::prepareToEnterSegment(){
  if(segment == preLanding) land();
  else if(segment == rightTaxiway) requestGate();
  else if(segment == rightTaxiway + 1 + gate) turnToGate();
  else if(segment == toGates + gate) atGate();
  else if(segment == fromGates + gate) backOnTaxiway();
  else if(segment == leftTaxiway) prepareToTakeOff();
  else if(segment == takeOff) depart();
  else ordinarySegment();
}
```

We will discuss the segment specific actions shortly. For now, note these actions may need resources and thus cause the plane activity to get suspeneded on. After such resources are acquired, the segment will be actually entered by calling member function `enter`, after which the movement on the segment will begin. Figure 28.3 shows this schematically.

```
void plane::enter(int newSegment){
  segment = newSegment;
  Position linestart = taxiways[segment]->getStart();
```

```
  moveTo(linestart.getX(), linestart.getY());
  Position lineend = taxiways[segment]->getEnd();
  face(lineend.getX(), lineend.getY());
  timeToSegmentEnd = taxiways[segment]->traversalT;
  sim::post(0, [=](){this->moveOnSegment();});
}
```

As you can see, the function visually positions the plane at the beginning of the line corresponding to the segment, and aligns to it. The other housekeeping tasks include setting `segment` to the index of the new segment on which the plane is to move, and initializing the counter `timeToSegmentEnd` to the time required to traverse this segment. Now the plane is ready to travel on the segment, and so this event is posted.

```
void plane::moveOnSegment(){
  if(timeToSegmentEnd != 0){
    if((segment == landing || segment == takeOff)
       && timeToSegmentEnd == taxiways[segment]->traversalT/2){
      taxiways[halfRW]->release();
    }
    forward(taxiways[segment]->stepsize);
    --timeToSegmentEnd;
    sim::post(1, [=](){moveOnSegment();});
  }
  else
    prepareToEnterSegment();
}
```

This function causes the plane to move on the taxiway segment `segment` on which it finds itself. Movement is possible so long as the plane has not reached the end, i.e. while `timeToSegmentEnd` is not 0. Till then we move forward by the stepsize determined for the current segment. However, if we are on the landing segment or the takeoff segment, we must also release `halfRW` when we pass the middle of these segments.

If the end of the segment has been reached, then the plane must attempt to enter the next segment. So we call `prepareToEnterSegment`.

We next discuss the segment specific actions needed to be performed before a segment can be entered.

28.5.1 Action `land`

This action is taken when the plane is on the (fictitious) `preLanding` segment. In order to land, the aircraft must acquire the landing runway, and then `halfRW` to satisfy half-runway exclusion as discussed earlier. After this, it may `enter` the landing runway.

```
void plane::land(){
  taxiways[landing]->acquire([=](){
    taxiways[halfRW]->acquire([=](){
      sim::log()<< "Plane " << id << " lands. scheduled arrival "
                << arrivalT << ", Service time " << serviceT << endl;
```

```
      show();
      enter(landing);
    });
  });
}
```

Note that on creation we have hidden the plane. The `show` command above makes it visible just as it is about to enter the runway.

28.5.2 Action `requestAGate`

This function is called when the plane is at the end of segment `rightTaxiway`. This is when a gate is to be allocated to the aircraft (Section 28.1.4).

The first part of the code above scans through the taxiway segments representing gates, and reserves the first one. Remember that the member `gate` was initialized negative, so that if it has become positive, then a gate must indeed have been allocated. If a gate is not allocated, we must try execute this function again, after one step. Thus, we will check every step till a gate can be reserved. This might seem inefficient, and the exercises ask you to find a better way.

After that the plane has to enter the next segment and move on it. So it acquires exclusive access to the segment. After getting exclusive access it releases the current segment and calls `enter`.

```
void plane::requestGate(){
  for(int i=0;i<nGates;++i)
    if (taxiways[toGates + i]->reserve()){
      gate = i;
      break;
    }
  if(gate < 0) sim::post(1,[=](){requestGate();});
                              // no gate? try again.
  else taxiways[segment+1]->acquire([=](){
    taxiways[segment]->release();
    enter(segment+1);
  });
}
```

28.5.3 Action `turnToGate`

This action is taken if the plane is at the end of segment `rightTaxiway + 1 + gate` (see Figure 28.2). This is precisely the segment from which we can turn to the gate `gate`, which is already allocated for the plane and where it wants to halt. The branch taxiway leading to gate `gate` is numbered `toGates + gate`. Thus, we should enter that segment. But before that we must release the current segment that we are on.

```
void plane::turnToGate(){
  taxiways[segment]->release();
  enter(toGates + gate);
}
```

28.5.4 Action `atGate`

This action is taken when the plane is on segment `toGates + gate`. This segment represents the gate allocated for the plane. It must wait here for its service time, and then start back towards the main path. The taxiway taking it back to the main path has index `fromGates + gate`.

```
void plane::atGate(){
  sim::log()<< " Plane " << id << " at gate " << gate
                << " will wait for " << serviceT << endl;
  sim::post(serviceT, [=](){     // wait for service
    enter(fromGates + gate);
  });
}
```

28.5.5 Action `backOnTaxiway`

This action is taken when the plane is at the end of segment `fromGates + gate`. This is a branch segment leading to main path segment `rightTaxiway + 2 + gate`. This is what the plane must enter on the way to take off. After it gets access, it must release the branch segment connecting to the gate, i.e. `togates + gate`. Then it should enter segment `rightTaxiway + 2 + gate`.

```
void plane::backOnTaxiway(){
  taxiways[rightTaxiway + 2 + gate]->acquire([=](){
    taxiways[toGates + gate]->release();
    enter(rightTaxiway + 2 + gate);
  });
}
```

28.5.6 Action `prepareToTakeOff`

This action is taken when the plane is at the end of the `leftTaxiway`. Before entering the take-off segment, i.e. the runway, we must acquire it, as well as acquire `halfRW`. Then we release the current segment and then `enter` the take-off segment.

```
void plane::prepareToTakeOff(){
  taxiways[takeOff]->acquire([=](){
    taxiways[halfRW]->acquire([=](){
      taxiways[segment]->release();
      enter(segment+1);
    });
  });
}
```

28.5.7 Action `depart`

The take-off is considered complete when the plane reaches the end of the take-off taxiway, i.e. the take-off runway. So we simply release the segment and hide the plane, i.e. make it invisible.

```
void plane::depart(){
  taxiways[segment]->release();
```

```
  hide();
  sim::log() << " Plane " << id << " left." << endl;
  delete this;
}
```

After this, we can return the plane object back to the heap. Notice that it is OK to call `delete this;` from within the member function, so long as you are not going to access `this` subsequently.

28.5.8 Action for an Ordinary Segment

Several segments don't require any special action. The function below deals with such segments. The next segment must be acquired, the current one released. Then we enter the next segment.

```
void plane::ordinarySegment(){
  taxiways[segment+1]->acquire([=](){
    taxiways[segment]->release();
    enter(segment+1);
  });
}
```

28.6 DEADLOCKS

A *deadlock* is a technical term used to describe a system in which one entity e_1 is waiting to reserve a resource held by entity e_2 which in turn is waiting to reserve a resource held by an entity e_3 and so on, till some entity e_n in this sequence is waiting to reserve a resource held by e_1. Notice that in this case no entity can make progress, because all are waiting for each other. As an example, Figure 28.4 shows cars deadlocked on roads in a city. Note that the roads are one ways, as shown. The cars in the top road are waiting for the space ahead of them to become empty. This will happen if the cars in the right road can move down. But these can move down if the cars in the bottom road can move left. These in turn will move left only if the cars in the left road can move. But these are blocked because of the cars in the top row. The net result: no one can move. Deadlock.

A deadlock is possible on a circular taxiway if every segment contains a plane which wants to move forward. In our airport, it would seem that the taxiways do not form a circular path. However we have to be careful in implementing the half-runway-exclusion rule.

It turns out that deadlocks will not arise because we observe the following discipline in reserving `halfRW`. A landing aircraft must first reserve the landing runway and only then `halfRW`. Similarly, a plane taking off must first reserve the takeoff runway and only then `halfRW`. You can see that this is a good strategy: `halfRW` being a precious resource must be reserved last. If a plane reserves `halfRW` and cannot reserve the landing runway, then it prevents take offs unnecessarily until such time as it reserves the landing runway. More formally, as the exercise asks you, you should be able to prove that with this policy there can be no deadlock. On the other hand, if landing planes as well as planes taking off reserve `halfRW` first, then it is possible to create a deadlock by carefully constructing the arrival sequence of the planes. The exercises invite you to explore this possibility.

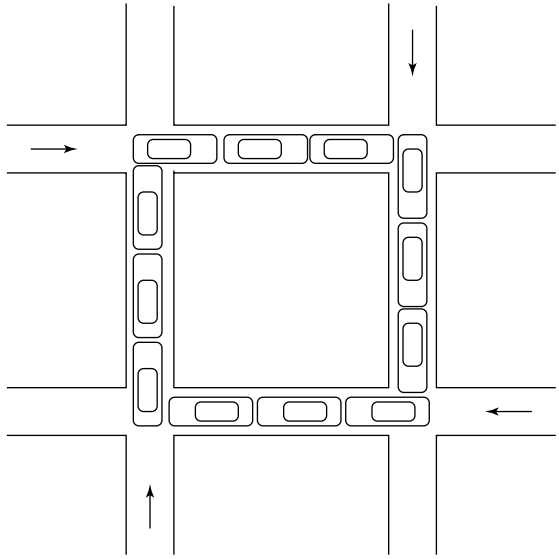


Fig. 28.4 *Traffic deadlock in a city*

28.7 CONCLUDING REMARKS

We have provided a somewhat substantial example of a discrete event simulation. It has several ingredients which are useful in general: for example, entities competing for resources.

The various member functions in the `plane` class effectively create a program that a plane is executing. However, the program is not given as single long text. Instead, it has to be given in pieces. This kind of style takes getting used to.

We could have designed this program differently: not use the `sim` and `Resource` classes at all, but write the code from the point of view of the air traffic controllers who examine the planes at each step and advance them as needed. You are asked to do this in an exercise.

EXERCISES

1. Modify the simulation program to print out the average aircraft delay.
2. Define a better `plane` class in which the on screen image looks like an aircraft rather than a triangle.
3. Suppose we wish to ensure that as much as possible, an aircraft must land at its arrival time. Thus, while granting `halfRW` to a departing plane, we must check whether no plane will want to land during the interval in which the departing plane will use `halfRW`. Device a good mechanism to do this. Hint: perhaps you can reserve the landing runway and `halfRW` a bit earlier than needed?
4. The program given in the text uses so called *busy waiting* to allocate gates, i.e. if a gate is not currently available, the plane retries after 1 step. It will be more efficient if the plane can `await` the release of *any* gate. Develop a class to represent such a resource group. A resource group models a sequence of objects, each of which can be either reserved or unreserved. On a `reserve` request, one of the unreserved objects must be allocated, i.e. the requesting entity should be set as

its owner. If all objects are currenly reserved, then the reserve request is deemed to fail and should thus return `false`. In that case, the entity may `await` its release. When any of the objects becomes available, that should get reserved for the waiting entity. Use this in the simulation code.

5. Show that our strategy of reserving resources ensures that there is no deadlock. Specifically, show that at every step some aircraft will make progress, and that there will not exist planes $p_0, \ldots, p_{n-1}$ where p_i is waiting for a resource held by p_{i+1} where $0 \leq i < n-1$, and p_{n-1} is waiting for a resource held by p_0.

6. Suppose we reserve `halfRW` first and then the take off or landing runways. Construct an input sequence (the file `arrivals.txt`) such that there is a deadlock.

7. Perform the airport simulation without using the `sim` and `Resource` classes, as described in Section 28.7. Do you think this will run faster than the simulation we have presented, or slower? What if there is no need to produce a graphical output, i.e. only a textual record is required?

8. Suppose we do not want to divide the taxiway into segments. Instead, suppose we will allow a plane to move a certain stepsize at each step while keeping a certain safe distance behind the plane ahead, if any. Implement this. The other rules must still be followed, i.e. the half-runway-exclusion rule and the rule that there can be at most one aircraft on each runway at any instant. Also, there can be only one aircraft on any branch taxiway.

9. Simulate an airport with two runways that do not intersect. Assume the same traffic as that for an airport with intersecting runways. Define the delay of an aircraft as the actual time it spends on the airport less the time it would spend if no other aircraft was present in the airport. Compute the total delay for all aircraft in both models. Increase the traffic i.e. arrivals per unit time and see how the total delay changes for the intersecting and non-intersecting runways.

10. If an aircraft is not allowed to land when it arrives at the airport, it must fly in a circular path of total duration some T and then try again. This is a more realistic model than what we have in the text. Simulate this model.

11. Consider the shortest path algorithm of Section 27.4. Suppose that we are also given the geometric coordinates for each vertex of the graph. Show a visual simulation of the algorithm, i.e. a turtle should move along each edge as if it were a cyclist.

CHAPTER 29

Systems of Non-linear Equations

Consider the following problem:

> A parallelepiped-shaped box has a base whose diagonal is 22 cm. The surface area of the box is 690 cm^2 and the volume is 1010 cm^3. What are the lengths of the sides of the box?

If x, y, z denote the side lengths in cm, clearly we have $x^2 + y^2 = 22^2$, $2(xy + yz + zx) = 690$ and $xyz = 1010$. This is a system of 3 equations in 3 unknowns. If the equations were linear in the unknowns, i.e. had the form $ax + by + cz = d$ where a, b, c, d are constants, then we could have solved them using the algorithm from Section 15.2.1. Unfortunately, these equations are non-linear. Thus, some new ideas will be needed.

Non-linear equations arise while modelling all but the simplest physical systems. Simple physical systems may give rise to linear equations, e.g. the electrical circuits we considered in Section 23.4. But the equations become non-linear, say if you add diodes as circuit elements (Exercise 9 of Chapter 8). Systems of non-linear equations arise in almost every area, from hydrodynamics and chemical equilibria to machine learning and game theory.

It turns out that solving non-linear simultaneous equations is much more difficult than solving linear systems. While there is no nice guaranteed method, there are some strategies which often work. Typically, these strategies are *iterative*, i.e. an approximate solution is somehow guessed, and then systematically improved. This is a very important paradigm in the numerical analysis of physical systems.

In this chapter, we will study an important member of the paradigm: the Newton–Raphson (NRM) method. We have already studied NRM in Section 8.4 for the one-dimensional case. Its generalization to multiple dimensions is in one way natural; but it is also trickier and grander. We will intuitively justify the multidimensional NRM, but a full mathematical treatment is outside the scope of this book.

After studying NRM for multiple dimensions, we will apply it to our box problem. Then we will consider the following problem in mechanics: given a chain of links of different lengths, compute the configuration in which it hangs if suspended from fixed pegs. We will see that NRM solves both

problems nicely. This chapter will thus serve as an introduction to the iterative paradigm as well as to a new set of applications.

29.1 NEWTON–RAPHSON METHOD IN MANY DIMENSIONS

In one dimension, NRM is used to find the root of a function f of one variable, i.e. find u such that $f(u) = 0$. The higher dimensional case is a natural generalization. We are now given n functions $f_1, \ldots, f_n$ each of n variables, and we want to find their *common* root, i.e. a set of values $u_1, \ldots, u_n$ such that $f_i(u_1, \ldots, u_n) = 0$ for all i.

As you might see, this is really the same as solving simultaneous, possibly non-linear equations. Any equation in n unknowns can be written so that the right hand side is 0, but then we can treat what is on the left hand side as a function of the unknowns. Indeed, our equations for the box problem can be stated in this form as follows:

$$f_1(x, y, z) = x^2 + y^2 - 484 = 0 \tag{29.1}$$

$$f_2(x, y, z) = 2(xy + yz + zx) - 690 = 0 \tag{29.2}$$

$$f_3(x, y, z) = xyz - 1010 = 0 \tag{29.3}$$

Indeed, the common roots x, y, z, of f_1, f_2, f_3 will precisely give us the side lengths of the box we want to construct.

We can think of each f_i as an *error function*, i.e. a function which we would like to be 0. Our goal is to find x, y, z such that each $f_i(x, y, z)$ becomes 0. So this suggests an idea. Suppose we have a current guess for the values of x, y, z. If all f_i evaluate to zero i.e. *there is no error*, we are done. Else we try to modify the current values so as to reduce the (absolute) value of each f_i. As you may remember from Section 8.4, NRM in one dimension proceeds in exactly this manner.

We will next present NRM in many dimensions. To keep the discussion simple, we will discuss the method in the context of the box problem, but as you will see, our discussion will be applicable to the general case. To get the method started, we need to find an initial guess for the values of the unknowns. This is not entirely trivial, and we will discuss this later. For now, suppose we somehow make an initial guess (x_0, y_0, z_0) for the side lengths of the box. In general we will have a guess (x_i, y_i, z_i), and will want to find the next guess $(x_{i+1}, y_{i+1}, z_{i+1})$.

The key idea in this is to generalize the tangent approximation, equation (8.4), of the one dimensional case. To begin with, let us just consider the first equation $f_1(x, y, z) = 0$. Consider what happens if we only vary x. In other words, we hold y, z fixed at y_i, z_i. But this is just the one dimensional case, and thus following equation (8.4) we can write $f_1(x_i + \Delta x) \approx f_1(x_i) + \Delta x \cdot f_1'(x_i)$. In this, we wrote $f_1'(x_i)$ to denote the derivative of f_1 with respect to x evaluated at $x = x_i$. However, since f_1 is a function of x, y, z, it is more appropriate to instead write the partial derivative of f_1 with respect to x, evaluated at (x_i, y_i, z_i). Thus, the approximation is

$$f_1(x_i + \Delta x, y_i, z_i) \approx f_1(x_i, y_i, z_i) + \frac{\partial f_1(x_i, y_i, z_i)}{\partial x} \Delta x \tag{29.4}$$

Next we keep x, z fixed, but allow y to vary. Reasoning exactly as above, we get

$$f_1(x_i + \Delta x, y_i + \Delta y, z_i) \approx f_1(x_i + \Delta x, y_i, z_i) + \frac{\partial f_1(x_i + \Delta x, y_i, z_i)}{\partial x} \Delta y$$

In this, for the first term on the right we will substitute the value from equation (29.4). For the second term we will use the approximation $\frac{\partial f_1(x_i + \Delta x, y_i, z_i)}{\partial x} \approx \frac{\partial f_1(x_i, y_i, z_i)}{\partial x}$ which is valid if Δx is small. Thus,

we get

$$f_1(x_i + \Delta x, y_i + \Delta, z_i) \approx f_1(x_i, y_i, z_i) + \frac{\partial f_1(x_i, y_i, z_i)}{\partial x}\Delta x + \frac{\partial f_1(x_i, y_i, z_i)}{\partial y}\Delta y$$

Proceeding as above, we next allow movement in the z direction, and get

$$\begin{aligned} &f_1(x_i + \Delta x, y_i + \Delta y, z_i + \Delta z) \approx \\ &f_1(x_i, y_i, z_i) + \frac{\partial f_1(x_i, y_i, z_i)}{\partial x}\Delta x + \frac{\partial f_1(x_i, y_i, z_i)}{\partial y}\Delta y + \frac{\partial f_1(x_i, y_i, z_i)}{\partial z}\Delta z \end{aligned} \tag{29.5}$$

Finally, we note that we would like to make f_1 zero, i.e. to set $f_1(x_i + \Delta x, y_i + \Delta y, z_i + \Delta z) = 0$. Thus, we should choose $\Delta x, \Delta y, \Delta z$ such that

$$0 = f_1(x_i, y_i, z_i) + \frac{\partial f_1(x_i, y_i, z_i)}{\partial x}\Delta x + \frac{\partial f_1(x_i, y_i, z_i)}{\partial y}\Delta y + \frac{\partial f_1(x_i, y_i, z_i)}{\partial z}\Delta z \tag{29.6}$$

Notice that this will not necessarily make $f(x_i + \Delta x, y_i + \Delta y, z_i + \Delta z)$ equal 0, because equation (29.5) which we started with was approximate. This is similar to NRM in one dimension: by going down the tangent we will usually not get to the root exactly. It will be convenient to write equation (29.6) slightly differently.

$$\frac{\partial f_1(x_i, y_i, z_i)}{\partial x}\Delta x + \frac{\partial f_1(x_i, y_i, z_i)}{\partial y}\Delta y + \frac{\partial f_1(x_i, y_i, z_i)}{\partial z}\Delta z = -f_1(x_i, y_i, z_i) \tag{29.7}$$

We can write similar equations for f_2, f_3.

$$\frac{\partial f_2(x_i, y_i, z_i)}{\partial x}\Delta x + \frac{\partial f_2(x_i, y_i, z_i)}{\partial y}\Delta y + \frac{\partial f_2(x_i, y_i, z_i)}{\partial z}\Delta z = -f_2(x_i, y_i, z_i) \tag{29.8}$$

$$\frac{\partial f_3(x_i, y_i, z_i)}{\partial x}\Delta x + \frac{\partial f_3(x_i, y_i, z_i)}{\partial y}\Delta y + \frac{\partial f_3(x_i, y_i, z_i)}{\partial z}\Delta z = -f_3(x_i, y_i, z_i) \tag{29.9}$$

Thus, we have 3 equations (29.7,29.8,29.9) in the 3 unknowns $\Delta x, \Delta y, \Delta z$. Note that they are linear: the coefficients are simply the partial derivatives evaluated at (x_i, y_i, z_i). Thus we can solve for $\Delta x, \Delta y, \Delta z$. From this we can get the next guesses $x_{i+1} = x_i + \Delta x, y_{i+1} = y_i + \Delta y, z_{i+1} = z_i + \Delta z$.

Let us work out the details for the specific functions in our box problem. Consider Eq. (29.7). The partial derivative of $f_1 = x^2 + y^2 - 484$ [Eq. (29.1)] with respect to x is $2x$, with respect to y is $2y$ and with respect to z is 0. The partial derivative of $f_2 = 2(xy + yz + zx) - 690$ with respect to x is $2y + 2z$, with respect to y is $2x + 2z$ and with respect to z is $2x + 2y$. The derivatives for $f_3 = xyz$ are yz, zx, xy. We merely need to evaluate these at (x_i, y_i, z_i) to get the equations.

We thus need a guess for (x_i, y_i, z_i). As you can see, our approximations work well if $\Delta x, \Delta y, \Delta z$ are small, i.e. our guess should have reasonably small error. So after some playing around, say we fix $x_i = 20, y_i = 10, z_i = 5$. So now we just need to substitute these into the expressions for the partial derivatives and the functions. Doing this, we get

$$40\Delta x + 20\Delta y = -16 \tag{29.10}$$

$$30\Delta x + 50\Delta y + 60\Delta z = -10 \tag{29.11}$$

$$50\Delta x + 100\Delta y + 200\Delta z = 10 \tag{29.12}$$

Solving this, we get $(\Delta x, \Delta y, \Delta z) = (-0.12, -0.56, 0.36)$. Adding these to (x_i, y_i, z_i) we get $(x_{i+1}, y_{i+1}, z_{i+1}) = (19.88, 9.44, 5.36)$. This guess is better: the new values of (f_1, f_2, f_3) are $(0.355215, -0.327948, 4.10383)$, which are much closer to zero than the old values $(16, 10, -10)$.

29.2 THE GENERAL CASE

In general, we have n equations $f_1 = 0, \ldots, f_n = 0$ in n variables $u_1, \ldots, u_n$. Let u_j^i denote the ith guess for u_j (i is *not* an exponent!). Let $\Delta u_j = u_j^{i+1} - u_j^i$ denote the change to u_j. Then we can proceed as above and get n linear equations, the kth of which is

$$\sum_j \frac{\partial f_k(u_1^i, \ldots, u_n^i)}{\partial u_j} \Delta u_j = -f_k(u_1^i, \ldots, u_n^i) \tag{29.13}$$

We solve these to get $\Delta u_1, \ldots, \Delta u_n$.

It is customary to consider the above equations in matrix form. Define an $n \times n$ matrix A^i in which $a_{kj}^i = \frac{\partial f_k(u_1^i, \ldots, u_n^i)}{\partial u_j}$. Define an n element vector b^i where $b_k^i = -f_k(u_1^i, \ldots, u_n^i)$. Let Δu denote the vector of unknowns $(\Delta u_1, \ldots, \Delta u_n)$. Then we can write the matrix equation

$$A^i \Delta u = b^i$$

in which A^i, b^i are known and we solve for Δu. The matrix A^i is said to be the Jacobian matrix for the problem. Further, defining $u^i = (u_1^i, \ldots, u_n^i)$ we get the update rule

$$u^{i+1} = u^i + \Delta u$$

Next we comment on when we should terminate the procedure, and how to make the first guess.

29.2.1 Termination

We should terminate the algorithm when all f_i are close to zero. A standard way of doing this is to require that $\sqrt{f_1^2 + \cdots + f_n^2}$ become smaller than some ϵ that we choose, say $\epsilon = 10^{-7}$ if we use `float`, and even smaller if we use `double` to represent our numbers. In keeping with our interpretation that f_i is the error, the quantity $(f_1, \ldots, f_n)$ is the vector error, and $\sqrt{f_1^2 + \cdots + f_n^2}$ is the 2-norm or the Euclidean length of the vector error.

The resulting method is given in Figure 29.1.

29.2.2 Initial Guess

Finding a good guess to start off the algorithm turns out to be tricky. In one dimension, we roughly plotted the function and sought a point close enough to the root. In multiple dimensions, this is more difficult.

Newton's method works beautifully if we are already close to the root. This is because very close to the root, the equations such as (29.7) become very accurate. One idea is to try to satisfy the equations approximately. It is often enough to satisfy only some of the equations. For example, we found for the box problem that a starting guess of $(x, y, z) = (20, 10, 5)$ worked quite well, and in two iterations got us a solution $(19.9085, 9.36214, 5.41884)$ which has error 2-norm about 10^{-4}.

On the other hand, an initial guess of (1,2,3) worked quite badly: it produced the "answer" (2.2659 -21.883 -20.3692). Note that this satisfies the equations closely, but surely we cannot have negative

Goal: Solve the system of n non-linear equations: $f_k(u_1, \ldots, u_n) = 0$, $k = 1, \ldots, n$.

1. Somehow guess initial values $u^0 = (u_1^0, \ldots, u_n^0)$. Set $i = 0$.
2. while $\sum_k f_k^2(u_1^i, \ldots, u_n^i) > \epsilon$ do
   a. Evaluate the Jacobian

$$A = \begin{bmatrix} \frac{\partial f_1}{\partial u_1} & \frac{\partial f_1}{\partial u_2} & \cdots & \frac{\partial f_1}{\partial u_n} \\ \frac{\partial f_2}{\partial u_1} & \frac{\partial f_2}{\partial u_2} & \cdots & \frac{\partial f_2}{\partial u_n} \\ \vdots & & & \\ \frac{\partial f_n}{\partial u_1} & \frac{\partial f_n}{\partial u_2} & \cdots & \frac{\partial f_n}{\partial u_n} \end{bmatrix}$$

   at $(u_1^i, \ldots, u_n^i)$. Let A^i denote the resulting matrix of values.
   b. Construct the vector b^i by setting $b_k^i = -f_k(u_1^i, \ldots, u_n^i)$
   c. Find the vector Δu by solving $A^i \Delta u = b^i$.
   d. Construct the new guess $u^{i+1} = u^i + \Delta u$.
   e. $i = i + 1$.
3. end while.
4. Print $u_1^i, \ldots, u_n^i$.

Fig. 29.1 *Newton–Raphson method in multiple dimensions*

side lengths! This points to another feature of non-linear equations: there may be multiple roots. Your iterative procedure may not necessarily take you to the correct one.

29.3 HOW A NECKLACE REPOSES

Suppose you are given a chain of n links, where the ith link has length L_i, $i = 0, \ldots, n-1$. Say the chain is hung from pegs at points (x_0, y_0) and (x_n, y_n) which are known. What is the shape attained by the chain when it comes to rest, hung in this manner? The links in the chain need not have equal lengths.

29.3.1 Formulation

Let x_i, y_i denote the coordinates of the left endpoint of link i, and x_{i+1}, y_{i+1} the coordinates of the right endpoint, where $i = 0, \ldots, n-1$. As discussed above, we already know the values of (x_0, y_0) and (x_n, y_n), these are the coordinates of the pegs from which the chain is suspended. The other x_i, y_i are the unknowns we must solve for. We are given the lengths L_i of the links, thus the variables x_i, y_i must satisfy the following equations.

$$(x_{i+1} - x_i)^2 + (y_{i+1} - y_i)^2 - L_i^2 = 0 \tag{29.14}$$

We must balance the forces and torques on the links. Suppose that F_i is the (vector) force exerted by link $i-1$ on link i (F_0 is the force exerted on link 0 by the left peg, and F_{n-1} is the force exerted by link $n-1$ on the right peg). Note of course that by Newton's third law, if one object exerts a force F on another, the latter exerts a force of $-F$ on the former. So each link i has a force F_i acting on

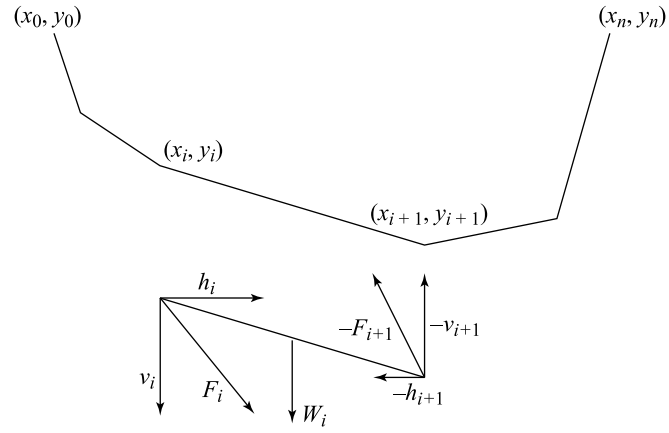


Fig. 29.2 *Forces on the ith link*

its left endpoint, and a force $-F_{i+1}$ acting on its right endpoint. Further, there is its weight, W_i, also vector, which acts at its center (Figure 29.2).

When the chain is at rest, total force on each link must be zero. Thus, for each link, we have $F_i - F_{i+1} + W_i = 0$. Suppose $F_i = (h_i, v_i)$, i.e. h_i and v_i are the horizontal and vertical components of F_i. Because W_i is only vertical, we can write it as $W_i = (0, w_i)$. The weight acts downwards, so perhaps we should write $-w_i$ as the y-component. However, do note that in the coordinate system of our graphics screen y increases downwards. Hence, we do not have the negative sign. Further, we will assume that the weight is proportional to the length, and so we will write $w_i = L_i$. Now balancing the horizontal component we get $h_i - h_{i+1} = 0$, i.e. all these variables are identical! Thus, we could write a common variable h instead of them. Balancing the vertical component we get, for all i:

$$v_i - v_{i+1} + L_i = 0 \tag{29.15}$$

We need to balance the torque as well. We do this around the right endpoint of the link. Remember that the torque due to a force F equals the magnitude of the force times the perpendicular distance from the center to the line of the force. The torque due to the horizontal component of F_i is simply the horizontal component times the vertical distance to the horizontal component. Thus, it is $h_i(y_{i+1} - y_i) = h(y_{i+1} - y_i)$. This torque is in the clockwise direction. The torque due to the vertical component is similar, $v_i(x_{i+1} - x_i)$, but in the counter clockwise direction. The distance to the line of the weight is $(x_{i+1} - x_i)/2$, and so the torque due to it is $L_i(x_{i+1} - x_i)/2$, also in the counterclockwise direction. But the total torque, considered in say the clockwise direction, must be zero. Thus, we get

$$h(y_{i+1} - y_i) - v_i(x_{i+1} - x_i) - L_i(x_{i+1} - x_i)/2 = 0 \tag{29.16}$$

Equations (29.14), (29.15), and (29.16) apply to each link, and thus we have $3n$ equations over all. The unknowns are $x_1, \ldots, x_{n-1}$ (noting that x_0, x_n are known) and similarly $y_1, \ldots, y_{n-1}$, and h, and $v_0, \ldots, v_n$. Thus, there are a total of $(n-1) + (n-1) + 1 + (n+1) = 3n$ unknowns. Thus, the number of unknowns and the number of equations match; however, our equations (29.14) and (29.16) are not linear. So we can use the Newton–Raphson method.

29.3.2 Initial Guess

Making a good initial guess is vital for this problem.

To make a good guess, we have to make use of our "common sense" expectation about what the solution is likely to look like. For the necklace problem, we can expect that the necklace will hang in the shape of a "U". So presumably, we can set (x_i, y_i) along a semicircle which hangs from the pegs. Also we can arrange the force values so that the total vertical force on each link is 0. One way to do this is to compute the total weight, and set v_0, v_n to bear half of it. Once we set this the other values of v_i can be set as per Eq. (29.15). The horizontal force h could be set to 0 to begin with. It is much trickier to try to balance the torque. But it turned out that the initial values as we have outlined here are enough to produce a good answer.

29.3.3 Experience

We coded up the algorithm and set the initial values as per the guessing strategy described above. After each iteration, we plotted the necklace configuration on our graphics screen. We found that the configuration quickly seems to reach a stable point. Indeed, we also printed out the error 2-norm, and it got close to zero fairly quickly.

29.4 REMARKS

You may observe that the NRM subsumes the linear case we discussed in Section 15.2.1. If each f_k is a linear function $f_k = p_{k1}u_1 + \ldots + p_{kn}u_n - q_k$, then the Jacobian will be a constant matrix with $A_{kj} = p_{kj}$. We will get an exact solution in 1 iteration of Figure 29.1 even with a starting guess of $u^0 = (0, \ldots, 0)$.

As you experiment with NRM, you might notice that the error norm (as defined in Section 29.2.1) does not necessarily decrease in each iteration. This is understandable, the error norm is guaranteed to decrease only if the equations such as (29.5) hold exactly.

Many methods have been suggested for solving non-linear equations. NRM is only one among them, and we discussed it to give you a flavour of what is possible.

EXERCISES

1. Write the program to solve the box problem.
2. Write the program to solve the chain-link problem. Display the configuration of the chain on the screen after each iteration of NRM.
3. In Exercise 9 of Chapter 8, you were asked to solve a circuit with a diode using the one dimensional NRM. Write a program to solve a circuit containing many branches as in Section 23.4, with several circuit elements being diodes.

APPENDIX A

Simplecpp

Simplecpp provides two macros, one function, one class for use in simulation, and several classes and functions for use in graphics. The functions and classes are placed in a namespace `simplecpp`.

We first discuss how to install Simplecpp. Then we give a very brief overview of the various features. Details are found in relevant parts of the rest of the book.

A.1 INSTALLATION

Simplecpp for Unix systems as well as Windows can be obtained by following instructions at

`www.cse.iitb.ac.in/~ranade/simplecpp`

`http://www.mhhe.com/ranade/cppp`

Your download will include the following directories:

1. `include` : C++ include files
2. `lib` : The library libsprite.a
3. `src` : Source files.
4. `programs_from_the_book` : Programs developed in the book. Ready to be compiled and executed.

A.2 NON-GRAPHICS FEATURES

The first Simplecpp primitive you will encounter in the book is `main_program`. This is a preprocessor macro (Section H), i.e. just a short form for `int main()`, which is how the main program is really required to begin in C++. Also see the discussion in Section 11.1 about this.

In Chapter 1, you will also encounter the primitive `repeat`. The looping statements in C++ such as `for` and `while` are more complex (Section 7). So in Simplecpp we designed the `repeat` primitive, which is very easy to understand, and can be used from the very first day. This is also implemented using a preprocessor macro (Section H).

Section 13.7.1 discusses the function `double randuv(double u, double v)`, which provides (pseudo) random numbers to be generated in the range `u` to `v`.

Chapter 27 discusses the classes `sim` and `Resource` which are useful for discrete event simulation.

A.3 GRAPHICS CLASSES AND FUNCTIONS

Chapter 1 describes the turtle-based graphics provided in Simplecpp.

Chapter 5 describes the more general graphics model, and the other graphics primitives. The primitive `getClick` using which a program can wait for the user to click on the canvas is also described.

Chapter 20 describes how a program can handle events such as pressing mouse buttons, dragging the mouse, and also pressing keys in the graphics mode.

A.4 COMPILING

When you install Simplecpp following directions, on Unix a script `s++` will get defined, using which you can compile. The simplest use is

```
s++ file-names-and-options
```

This invokes the Gnu C++ compiler using the command `g++`. We have included many options already with `s++`. Some of our options specify where the Simplecpp library and include files can be found. We have also specified several options to produce warnings by default. But you can give even further options on the command line provided they are supported by `g++`. One option which you might wish to give is `-o executable-file-name`. If this option is not specified, `g++` calls the executable `a.out`, which is what we have been using in the book.

APPENDIX B

Reserved Words in C++

The following words are reserved for special use by the C++ language and cannot be used as identifiers.

alignas	alignof	and	and_eq	asm	auto
bitand	bitor	bool	break	case	catch
char	char16_t	char32_t	class	compl	const
const_cast	constexpr	continue	decltype	default	delete
do	double	dynamic_cast	else	enum	explicit
export	extern	false	float	for	friend
goto	if	inline	int	long	mutable
namespace	new	noexcept	not	not_eq	nullptr
operator	or	or_eq	private	protected	public
register	reinterpret_cast	return	short	signed	sizeof
static	static_assert	static_cast	struct	switch	template
this	thread_local	throw	true	try	typedef
typeid	typename	union	unsigned	using	virtual
void	volatile	wchar_t	while	xor	xor_eq

Words such as `main`, `cin`, `cout` are not reserved. However, it is strongly recommended that you do not use them for anything other than their commonly accepted purpose.

APPENDIX C

Operators and Operator Overloading

We will discuss some of the less frequently used operators, and then consider operator overloading.

C.1 BITWISE LOGICAL OPERATORS

C++ allows bitwise logical operations to be performed on variables of integer types. For simplicity, we will only consider operations on the `unsigned` types.

C.1.1 OR

The operator `|` is the bitwise OR operator, i.e. `p | q` returns a number that is obtained by taking the binary representations of `p, q`, and computing the OR of the corresponding bits. Note that the logical OR of two bits is a 1 if and only if at least one of the bits is 1. Here is an example.

```
unsigned int p=10, q=6, r;
r = p | q;
```

This would cause the bit pattern

00000000000000000000000000001010

for 10 (decimal) to be OR'ed with the bit pattern

00000000000000000000000000000110

for 6, to get the result

00000000000000000000000000001110

which is the bit pattern for 14. Thus, at the end, `r` would be 14.

C.1.2 AND

The operator `&` performs bitwise logical AND. Note that the logical AND of two bits is 1 iff both bits are 1. Thus, for `p, q` as defined above, if we write

```
unsigned int s = p & q;
```

`s` would get the bit pattern 00000000000000000000000000000010, which is the bit pattern for 2. Thus at the end, `s` would equal 2.

C.1.3 Exclusive OR

The operator `^` is the bitwise exclusive OR operator. Note that the logical exclusive OR of two bits is 1 if and only if exactly one of the bits is 1. Thus, if we write

```
unsigned int t = p ^ q;
```

the variable `t` would get the bit pattern 00000000000000000000000000001100, which is the bit pattern for 12. Thus, `t` would be 12 after the statement.

C.1.4 Complement

Finally, the operator `~` is the (unary) bitwise complement operator. The complement of a bit is 1 if and only if the bit is 0. Thus, if we write

```
unsigned int u = ~p;
```

the bit pattern for `u` would have 0s wherever `p` had 1s and vice versa. Thus, we would have 1s in all positions except the positions of place value 2 and 8. Thus, the value of `u` after the statement would be $(\sum_{i=0}^{31} 2^i) - 2 - 8 = 4294967285$.

C.1.5 Left Shift

The operator << is used to shift a bit pattern to the left. Thus, `x << y` would cause the bit pattern for `x` to be shifted left by the the value of `y`. By this, we mean the following operation. We first throw away the most significant `y` bits, and then append y zeros on the right. The resulting bit pattern is the result of the operation. Note that if the most significant `x` are zero, then `x << y` is simply $x2^y$, where x, y are the values of `x` and `y` respectively.

For `p, q` as we defined, i.e. having values respectively `10` and 6, if we write

```
unsigned int v = p << q;
```

the variable `v` would get the bit pattern 00000000000000000000001010000000. This has the numerical value $10 \times 2^6 = 640$.

C.1.6 Right Shift

The operator >> is used to shift a bit pattern to the right. Thus, `x >> y` would cause the bit pattern for `x` to be shifted right by the the value of `y`. By this we mean the following operation. We first throw away the least significant `y` bits, and then append y zeros on the left. The resulting bit pattern is the result of the operation. Note that `x >> y` is simply $x/2^y$ (integer division), where x, y are the values of `x` and `y` respectively.

For `v` as we defined above, i.e. having value respectively `640`, if we write

```
unsigned int w = v >> 2;
```

the variable `w` would get the bit pattern 00000000000000000000010100000. This has the numerical value 160.

C.2 COMMA OPERATOR

A comma expression has the form

```
lhs, rhs
```

where `lhs` and `rhs` are expressions. The operator causes the evaluation of both the expressions, and the value of `rhs` is used as the result of the comma expression.

The comma operator can be used to force the evaluation of multiple expressions in settings where syntactically only one expression is expected.

A common use is to have multiple increment and decrement operations in a `for` statement.

```
for (int y = 10, x=0; y>0; y--, x++)
  cout << x << endl;
```

This would print the numbers 0 through 9.

As another example, using the comma operator, our mark averaging program of Section 7.1.2 can be written more compactly as follows.

```
main_program{
  float nextmark, sum=0;
  int count=0;

  while(cin >> nextmark, nextmark >= 0){
    sum = sum + nextmark;
    count = count + 1;
  }

  cout << "The average is: " << sum/count << endl;
}
```

However, usage such as above is not common, and hence is not recommended.

Also note that the comma operator should not be confused with the use of the comma as a delimiter in declarations e.g. `float nextmark, sum=0;` above, or function calls, e.g. `f(a,b)`.

C.3 OPERATOR OVERLOADING

We have discussed the basic ideas of operator overloading in Section 18.4 and Section 18.11.1. Here we discuss some details.

The following prefix unary operators can be overloaded

```
+ - * & ! ~ ++ --
```

For any operator `@` in the list above, overloading can be done either by defining a member function `operator@` in the class of the operand taking no argument, or by defining a non-member function `operator@` taking a single argument of the type of the operand.

The unary suffix operators `++, --` can also be overloaded. You again define a member or ordinary function `operator@` like the prefix versions. However, to distinguish from the prefix versions, you also have an extra `int` argument which you ignore. This might seem arbitrary, and it could indeed be considered a *hack*.

APPENDIX D

Formatted Output

While printing out data on the screen, or outputting it into a file (Section 18.9), you may wish to control its appearance in different ways. For example, you may want exactly 2 digits to appear after the decimal point while printing a number. Or you might want names or numbers in different lines aligned at the same column. It is possible to do such things by using *manipulators*, which we discuss in this appendix.

A manipulator can be an identifier or an identifier with a parameter, like a function call. To use the manipulators that take a parameter you must include the header file `<iomanip>`. A manipulator must be inserted into the output stream using the << operator just as data is inserted. We will see examples shortly.

Our examples are all for the standard output stream, `cout`. However, the discussion applies to all output streams.

D.1 GENERAL-PURPOSE MANIPULATORS

The identifier `endl` that you have been using is actually a manipulator. It inserts the newline character into the output stream and then flushes the stream, i.e. causes it to be printed immediately. There is also a `flush` manipulator which flushes the stream without inserting a newline.

An important general-purpose manipulator is `setw`. This has an integer parameter which specifies the width, or the minimum number of characters to be used for printing the next output value. If the output value needs fewer characters than the specified width, then *fill* characters are added to make up the width. The fill character is a space by default. By default, the fill characters are added to the left, i.e. by default the value is right justified in its field. Here is an example.

```
cout << '*' << setw(10) << 56 <<'*'<< endl;
```

In this we have put in `'*'` merely to make it obvious where the spaces have been inserted. This will cause the following to be printed.

```
*        56*
```

The manipulator `setw` is non-persistent, i.e. its effect is limited to just one value following it. Other manipulators that we see next are persistent, i.e. they will apply to all future values to be printed.

The `left` and `right` manipulators enable you to get left/right justification. Thus, we might write

```
cout << '*' << setw(10) << left << 56 <<'*'<< endl;
```

to get

```
*56        *
```

Field width and justification also applies to character strings.

```
cout << '*' << setw(10) << left << "xyz" <<'*'<< endl;
```

will produce

```
*xyz       *
```

The `setfill` manipulator takes a character as argument and uses that to fill.

```
cout << setw(10) << right << setfill('-') << "xyz" << endl;
```

will produce

```
-------xyz
```

D.2 NUMERIC DATA

In accounting applications, it is often desired that the sign of a number be justified to the left, while the number is justified to the right, in the specified width. This can be accomplished using the `internal` manipulator.

```
cout << '*' << setw(10) << internal << -3.14 <<'*'<< endl;
```

will produce

```
*-     3.14*
```

Normally, positive numbers are not prefixed with +. But you can get that to happen using the manipulator `showpos`. The manipulator `noshowpos` will inhibit showing +.

Next we consider the printing of floating point numbers. There are many choices. As you might have observed, if the number is small, the default is to print it in the usual fixed point manner, with a decimal point if needed. If the number is large, then the scientific notation is used. Further, the number of digits of precision to be shown must also be decided. There are some defaults; however, they can all be changed.

The manipulators `fixed` and `scientific` respectively print a number using the fixed point notation and scientific notation.

```
cout << fixed << 6.023e23 << ' ' << scientific << 3.14 << endl;
```

will produce

```
602299999999999975882752.000000 3.140000e+00
```

You might have expected 6.023e23 to be printed as 602300000000000000000000. What you see is slightly different because 6.023e23 is first converted to `double` which introduces some error.

The manipulators `scientific` and `fixed` are persistent. So if you want to return to the default situation in which C++ decides which notation to use depending upon the magnitude of the number being printed, you must issue the complicated looking command

```
cout.unsetf(ios::fixed | ios::scientific);
```

The maximum number n of digits of precision can be controlled by using the manipulator setprecision(n). The phrase "digits of precision" includes the number of digits before and after the decimal point.

Normally, the number of digits printed after the decimal point depend upon how many are needed. However, you may wish that a certain number be always printed, say in order to align data. You can accomplish this using the manipulator showpoint.

```
cout << setprecision(5) << 3.1415926535 << endl << 87.25 << endl;
cout << showpoint << 3.1415926535 << endl << 87.25 << endl;
```

This will produce

```
3.1416
87.25
3.1416
87.250
```

After this, you can revert to not printing trailing zeroes by using the manipulator nowshowpoint if you wish.

D.2.1 Base

You can change the base to which a number is printed by specifying the manipulators dec, hex and oct, which cause subsequent numbers to be printed using the decimal, hexadecimal and octal number systems. Note however, that floating point numbers are always printed in decimal. Further, while printing negative integers in hexadecimal or octal, the corresponding bit patterns are considered and printed without a sign. Simply put, it makes sense to only print unsigned integers in either hexadecimal or octal.

It is possible to give an indication of the base used in printing, using the manipulator showbase. This will prefix hexadecimal values with an x, and octal values with a leading 0. The leading indicators can be suppressed using the manipulator noshowbase.

D.2.2 Capitalization

If you really care, you can have the radix indicators as well as the 'e' in scientific notation appear capitalized, using the manipulator uppercase. You can get back to the default, lowercase, by the manipulator nouppercase.

D.3 BOOLEAN DATA

You can make boolean data appear as true or false by using the manipulator boolalpha, and as 1 or 0 (default) using the manipulator noboolalpha.

APPENDIX E

The stringstream Class

The class iostream is used to define objects such as cin and cout on which we can use the extraction operators >> and << respectively to read or write data. The objects are called streams, because data flows in and out of them.

A stringstream is a stream object, but it is constructed out of a string. To use it, you need to include the header <sstream>. This is especially useful in extracting numbers from strings or converting numbers to strings.

As an example, here is a program that takes two double numbers as command line arguments and prints their product.

```
#include <sstream>
int main(int argc, char *argv[]){
  double x,y;
  stringstream(argv[1]) >> x;
  stringstream(argv[2]) >> y;
  cout << x*y << endl;
}
```

In this, we have used the stringstream functionality provided in C++, by including <sstream>. The function stringstream takes a single argument s which is a character string, and converts it to an input stream (such as cin). Now we can use the » operator to extract elements. Thus, stringstream(argv[1]) » x; would extract a double value from the second word typed on the command line. Similarly, a double value would be extracted into y from the third word. Thus, if you typed

```
a.out 4 5e3
```

The answer, 20000 would indeed be printed.

Here is another example.

```
int main(){
  string s="1 2 3";
  int x,y,z;
  stringstream(s) >> x >> y >> z;
```

```
  stringstream t;
  t << x*y <<' '<< y*z;
  cout << t.str() << endl;
}
```

As you can see, in this we have made multiple extractions from the same `stringstream`. This is allowed. Basically, everything that you can do with streams is allowed on stringstreams. The stringstream `t` is used for output, and we have put multiple values into it. Finally, the member function `str` allows us to extract the `string` out of a `stringstream`, which can be printed out if desired.

APPENDIX F

Exceptions

How to deal with unexpected events is a ticklish problem while writing programs. By "unexpected event" we mean something like: you request memory using the operator `new` but for some reason the request is not granted. Another unexpected event could be that the user typed a non numerical value in response to a request to type in a number. Another possibility is that your program is trying to read from a file, but it finds that the file cannot be read because of problems with the disk. Another possibility is that an argument specified to a function does not satisfy the preconditions, e.g. the user specified negative numbers as arguments to a function to find the greatest common divisor.

You may not wish to write code to respond to such "rare" events in your program; you may consider it fine if the program just aborts. Or you may wish to handle such events and try to recover from them. For example, if there is a problem with a file you may ask the user to supply another file. How should such code be organized? Should you check for exceptional events after every file reading operation? If you do, it will likely clutter up your program. It is nicer if you can organize your code into two parts: the code which handles the normal situation, and the code that handles the exceptional events. The key point is that in the code that handles the normal events, you don't even want to write checks for exceptional events, let alone describe how to handle them. The exception-handling mechanism of C++ allows you to do precisely this.

The first important provision is a way by which a function can choose to exit *abnormally*. Instead of executing the usual `return` statement, a function can execute a `throw` statement. If a `throw` statement is executed, the function call terminates, and instead of resuming from the point of the call, the control executes a *handler* which must have been defined earlier. The handler can take appropriate actions and continue if possible. Here is an example of a `throw`.

```
int gcd(int m, int n){
  if(n <= 0) throw "Non-positive argument to gcd.";
  while(n > 0){
    int t = m % n;
    m = n;
    n = t;
  }
  return m;
}
```

The handler is defined using the `try` and `catch` statements as shown below.

```
int main(){
  try{
    cout << gcd(36,24) << endl;
    cout << gcd(36,0) << endl;
    cout <<"Done!\n";
  }
  catch(...){
    cout <<"Probably gcd was supplied a 0 argument." << endl;
  }
  cout << gcd(153,68) << endl;
  cout << gcd(3,0) << endl;
  cout <<"Done!!\n";
}
```

In this, the code inside the `try` block executes like normal code, except that if some call resulting from that execution throws an exception (which simply means executing a `throw` statement), then the execution continues from the `catch` statement. Thus, in this case, after the call `gcd(36,0)`, the `throw` statement will be executed. Thus, `"Done!"` would not be printed, but the `catch` block will be executed, and the message `"Probably .."` would be printed. After executing all the statements in the `catch` block, the execution of the `try-catch` group would end. After that the statement following the group will be executed.

The next statement is a call to calculate the GCD of 153, 68. This would cause 17 to be printed. The call after that requests the GCD of 3 and 0. In response to this an exception would be thrown. This time the call is not inside a `try` block, so there is no `catch` block to continue from. In this case, the program would abort. Thus the final `"Done!!"` would not be printed.

Finally, note that if no exception is thrown inside a `try` block, then the execution of the `try-catch` group would complete after the execution of the code in the `try` block.

In the catch statement above, we did not make use of the message following the `throw` statement in `gcd`. We will discuss how to use the message next.

F.1 THE GENERAL FORM

In general, a `throw` statement can return any object. In the above example, we chose to return a `const char*`, but we could return other objects too.

The ellipsis in the form `catch(...)` indicate that the block following it should be executed for all throw statements. In this form, the code in the block does not get access to the object being returned by the `throw`.

Alternately, you could use the form`catch`

```
catch(param-type param){
code
}
```

Such a `catch` block would be executed if the type of the object thrown matches `param-type`. In such a case, `param` will denote the object thrown. Thus, to be able to access the object returned by `gcd` the `catch` statement should have been

```
catch(const char* m){
  cout << m << endl;
}
```

With this, we would have the message in the object thrown by `gcd` printed out.

Note that you can have several `catch` blocks, each catering to thrown objects of different types. If the type of the object thrown does not match any of the types given in the different catch blocks, then the program will abort.

F.2 ENABLING INPUT EXCEPTIONS

We have mentioned earlier that if the user types a non-numerical value in response to a request to type a number, the program does *not* abort but just continues. Furthermore, the behaviour of the program is unpredictable from that point onwards. However, if an error has occurred, the stream becomes `NULL`, as we observed in Section 13.6.2. So we could check whether a stream has become `NULL` after every input operation.

However, you may find this cumbersome. You may wish: why does the program not abort automatically if a bad value is read? You can indeed arrange for this.

You can request that a stream should throw an exception when an input operation fails, say because the type of the character read is not of the type required. This can be done by writing:

```
streamname.exceptions(std::ifstream::failbit|std::ifstream::badbit);
```

In this, `streamname` is the name of the stream from which you are reading. The arguments to the command are constants defined in the header file `<fstream>`. So you should include that file.

The above line will cause an exception to be thrown on file reading errors. If you do not catch the exception, it will cause the program to abort. This may just be what you need most of the time.

You may of course, choose to handle that exception. Perhaps you want the user to type in the input again. For this, you can put the reading statements inside a `try` block and have a `catch` block in which the recovery code is put.

The type of the object thrown on file-reading errors is `exception`, which is defined in the header file `<exception>`. Thus, you can write a catch block

```
catch (exception& e){ cout << e.what() << endl; }
```

The class `exception` has the member function `what` which is used to describe the reason for the exception. The catch block can contain other code too, e.g. if you want the user to type the value again.

APPENDIX G

Managing Heap Memory

Dynamic allocation of heap memory is required for many purposes, e.g., representing variable-length entities. The simplest way of doing this is to use standard library classes (Chapter 22). The standard library classes hide the memory management behind a simple convenient interface which is often adequate. But in some applications, e.g. formula layout (Chapter 24), it is necessary to build your own classes in which memory management must be done. In Chapter 21, we discussed a simple strategy for managing heap memory. Here we will discuss a more sophisticated strategy. This strategy is based on the `shared_ptr` and `weak_ptr` classes available on including the header file `<memory>`.

The simple memory-management strategy presented in Section 21.3.1 was based on a "no sharing" principle: we ensured that each allocated object is pointed to by exactly one pointer. This principle enables us to fairly easily decide when an object allocated on the heap is no longer needed by the program. When this is determined, the memory given for the object can be returned to the heap, as is done in the various member functions developed in Section 21.3. This no sharing strategy is common, and is used (behind the scenes!) in the standard library.

However, the constraint that each object be pointed to by at most one pointer is not always efficient or convenient. As a simple example, consider the `String` class of Figure 21.2. Suppose we write the following code.

```
String x,y;
x = "ABC";
y = x;
```

As per the implementation of Figure 21.2, the second statement would allocate a character array on the heap, and `x.ptr` would point to it. The third statement will allocate another array into which the previous array would be copied. Then it would set `y.ptr` to point to the new array. This is consistent with the no sharing principle. But it seems we could do better. Do we really need to make a copy? Why not just assign `y.ptr = x.ptr;` and have both pointers point to the same array in memory? This would save on memory requirement and also on copying time. However, as we pointed out in Section 21.3.1, having `x, y` share a character array allocated on the heap creates problems too. Basically, it becomes harder to detect when the array is not needed and can be returned back. For example, if the above code was followed by

```
y = "DEF";
x = "12";
```

then clearly the array containing "ABC" is no longer needed, and that memory should be returned back using delete []. The key question is: how do we know whether a particular array is not pointed to by any pointer and is thus not in use? The solution we consider here uses *reference counting*. This is already available to you in C++ through the header file <memory>.

G.1 REFERENCE COUNTING

The basic idea is to associate a *reference count* with each object allocated on the heap, say with the character array containing "ABC" that was discussed above. The reference count of an object is simply the number of pointers pointing to that object, indicating how useful that object is. Typically, the reference count is stored in an auxiliary variable. If we create a an additional pointer to point to an object, we must add 1 to its reference count. If we remove a pointer, then we must subtract 1. When the count of an object X drops to zero, we decide that X is no longer useful, and so we must return the memory of X to the heap. Note that X might itself contain a pointer to an object Y. In this case we know that the pointer to Y out of X will no longer exist. So the reference count of Y must also be decremented. If this causes the reference count of Y to drop to zero, Y must also be returned back to the heap, and so on.

Using reference counts, we can indeed allow character arrays to be shared between String objects as discussed above. For this, we would need to have auxiliary variables to store the reference counts, and write the code to increment and decrement them appropriately. Such code is a bit tedious, and so C++ provides a class shared_ptr which does all this for you internally. We will see its use shortly.

It turns out that reference counting is not adequate when objects can point to each other, directly or indirectly. We will consider this in Section G.3. A solution will also be discussed. This is based on another class, weak_ptr.

To use shared_ptr or weak_ptr, you need to include the header file <memory>.

G.2 THE TEMPLATE CLASS shared_ptr

A shared_ptr is really a small structure that contains the real pointer, and other data needed to manipulate reference counts. The constructor, copy constructor and the assignment operator are defined to perform the respective functions and in addition modify the reference count as described above. Specifically, as these member functions execute, the reference count will increase or decrease suitably, and if it drops to zero, the pointed object will be deleted. Also, if a shared_ptr is itself deleted, or goes out of scope, the reference count of the object pointed to must decrease. This will happen because the destructor of shared_ptr is defined to do so. The dereferencing operator * is also defined for a shared_ptr, and it causes the object pointed to to be returned.

As you can see, a shared_ptr is like an ordinary pointer in many ways, and can be used almost as conveniently. The operations needed to maintain the reference count happen behind the scenes! Note, however, when we speak of a reference count above we mean the number of shared_ptrs pointing to an object. If an object has ordinary pointers pointing to it, then the reference count as defined above will not count them. Informally stated, if you want reference counting to work for a certain object, you must only point to it through shared_ptrs.

Some additional member functions are also provided. The member function `use_count` returns the reference count of what the `shared_ptr` points to. The member function `get` returns an ordinary pointer to the object pointed. This is useful for performing indexing operations as you will see.

G.2.1 Shared Pointers in the `String` Class

We now show how to modify the implementation of the `String` class of Section 21.3. Our modification will enable sharing of values between two `String` objects, as discussed in the introduction. The new implementation is shown in Figure G.1. It uses the functions `length` and `scopy` from Chapter 21.

You may find it interesting that this code is shorter than the code of Figure 21.2. This because there are no constructors and only one assignment operator. As you will see, the default versions of others will suffice.

We will use this with a main program shown in Figure G.2. When you execute this program, in the very first statement, variable `a` is created. The default constuctor is used for this. This constructor will in turn create `a.ptr` using the default constructor for `shared_ptr`. Thus effectively the pointers will be set to `NULL` and counts to 0. The next statement stores `"PQR"` in `a`. After this we see that the reference count of `a` becomes 1.

After that variable `b` is created, initialized to `a`. For this, the default copy constructor gets used,

```
class String{
  shared_ptr<char> ptr;
public:
  String& operator=(const char* rhs){
    ptr = shared_ptr<char>(new char[length(rhs) + 1]);
    scopy(ptr.get(),rhs);
    return *this;
  }
  String operator+(const String &rhs) const {
    String res;
    res.ptr = shared_ptr<char>(new char[size() + rhs.size() + 1]);
    scopy(res.ptr.get(), ptr.get());
    scopy(res.ptr.get(), rhs.ptr.get(), length(ptr));
    return res;
  }
  void print() const {
    if(ptr != NULL) cout << ptr << endl;
    else cout << "NULL" << endl;
  }
  int size() const {return length(ptr.get());}
  char& operator[](int i) const {return ptr.get()[i];}
  int use_count() const {return ptr.use_count();}
};
```

Fig. G.1 `String` *with value sharing*

```
int main(){
  String a;
  cout <<"a use counts: "<<a.use_count()<<endl;  a.print();
  a = "PQR";
  cout <<"a use counts: "<<a.use_count()<<endl;  a.print();

  String b(a);
  cout <<"ab use counts: "<<a.use_count()<<b.use_count()<<endl;
  a.print(); b.print();

  a = "ABC";
  cout <<"Overwrote a. ab use counts: "<<a.use_count()
  <<b.use_count()<<endl;
  a.print(); b.print();

  b = a;
  cout <<"b=a. ab use counts: "<<a.use_count()<<b.use_count()<<endl;
  a.print(); b.print();
}
```

Fig. G.2 *Main program for testing*

which is adequate. This, in turn, causes the copy constructor to be invoked on `b.ptr` using `a.ptr`. Thus, we will get `b` also to point to `"PQR"`. Thus the reference count for `a.ptr` and `b.ptr` will both become 2.

Next we store `"ABC"` in `a`. The print statements show that `b` continues to have value `"PQR"`, but the reference counts have both dropped to 1. This is as expected.

Finally, when we copy `a` to `b`, the default assignment operator gets used. This does a member by member copy. Thus `a.ptr` is assigned to `b.ptr`. This assignment causes the real pointers inside the `shared_ptr` to be copied, and also the reference counts to be updated. This is seen in what is printed.

G.2.2 General Strategy

Suppose you wish to manage allocation of a certain class `X`. The preceding discussion suggests the following strategy: hold all pointers to heap allocated objects of class `X` only in `shared_ptr`s. Allocate memory as always using `new`, but store the resulting pointer in a `shared_ptr`. If you do this, you will not have to worry about deallocating memory. This will happen automatically when the reference counts go to 0.

This strategy works well sometimes. It worked well for the `String` class above. We will see that it will work well also for manipulating expression trees.

However, any implementation of reference counting (including that provided by `shared_ptr`) has one fundamental limitation: if your pointers form cycles, then you will have memory leaks even with `shared_ptr`s. We will see this in Section G.3.

G.2.3 Shared Pointers in Expression Tees

We develop the code for implementing expression trees (Section 24.2.2) that returns memory to the heap when not needed. The code is shown in Figure G.3.

```
class Exp;
typedef shared_ptr<Exp> spE;

class Exp{
  spE lhs, rhs;
  string value;
  char op;
public:
  Exp(string s): value(s), op('P') {cout << "Created "<< s<< endl;}
  Exp(char o, spE l, spE r): lhs(l), rhs(r), op(o) {
    cout << "Created exp with op "<< o << endl;
  }
  ~Exp(){
    if(op == 'P') cout <<"Deleting "<<value<<endl;
    else cout <<"Deleting exp with op "<< op << endl;
  }
};
spE literal(string s){return spE(new Exp(s));}
spE formula(char o, spE l, spE r){return spE(new Exp(o,l,r));}

int main(){
  spE e1 = literal("100");                  cout <<"e1 created.\n";
  spE e2 = formula('+', e1, literal("200"));cout <<"e2 created.\n";
  spE e3 = formula('*', e2, e1);            cout <<"e3 created.\n";
  e2 = literal("300");                      cout <<"e2 modified.\n";
  e3 = e1;                                  cout <<"e3 modified.\n";
}
```

Fig. G.3 *Expression trees with memory management*

We use the class `Exp` to represent expressions. In the code, it is necessary to form `shared_ptr`s to `Exp` quite frequently. So we have a type definition, `spE`, shared pointer to expressions. In the main program, it is convenient to think of formulae as being of type `spE`.

The main program uses functions `literal` to create literals, i.e. primitive expressions. The function `formula` is used to create non-primitive expressions. As you can see, the functions allocates memory using `new`, as needed. However, there is no explicit code to return memory back to the heap. This happens automatically through the `shared_ptr` code.

We consider what happens when the main program is executed. After the first line, the following gets printed.

```
Created 100
e1 created.
```

When the second line is executed, the following is printed.

```
Created 200
Created exp with op +
e2 created.
```

The key point to note here is that the literal 100 is *not* created again, i.e. it is shared with `e1`. The execution of the third line gives

```
Created exp with op *
e3 created.
```

The notable point here is that only the top node is created, the rest is shared. The fourth line prints the following:

```
Created 300
e2 modified.
```

Nothing is yet destroyed, because all the nodes allocated thus far are in use. The fifth line prints the following.

```
Deleting exp with op *
Deleting exp with op +
Deleting 200
e3 modified.
```

When `e3` is assigned, the top node of `e3`, i.e. the one with operator `*` is no longer needed. Its reference count goes to zero, and hence it is deleted. But its destructor will cause the destructor to be called on its members `lhs` and `rhs`. The left-hand side, `lhs`, is no longer needed, and it would be reflected in its reference count. Thus that is also deleted. This also causes the literal 200 to be deleted.

After that the program ends. This causes the variables `e1, e2, e3` to be deleted, because control leaves the main scope. At this point `e3, e1` have the value 100, and `e2` has the value 300. so these nodes are also deleted, giving the messages:

```
Deleting 300
Deleting 100
```

The various messages printed should pursuade you that memory is indeed being shared and returned when not needed.

You may want to modify the program by inserting calls to member function `use_count` to find out for yourself what happens to the use count as the program executes.

G.3 WEAK POINTERS

Consider the following program.

```
struct A{
  shared_ptr<A> ptr;
  A(){cout << "Creating "<< this << endl;}
  ~A(){cout << "Deleting "<< this << endl;}
};
int main(){
  shared_ptr<A> s1(new A), s2(new A);
  s1->ptr = s2;
  s2->ptr = s1;
  s1 = NULL;
```

```
  s2 = NULL;
}
```

When you execute the program, it will print

```
Creating 0x804c008
Creating 0x804c030
```

There will be no deallocations and hence n `"Deleting ..."`.

Let us understand why this happens. The program begins by creating the shared pointers `s1, s2` to freshly allocated structures. From the messages printed, we know that these structures are at `0x804c008` and `0x804c030`. After step 1 of the program, these structures have use counts 2. When we execute `s1->ptr = s2` and `s2->ptr = s1`, the structures `0x804c008` and `0x804c030` get an additional pointer, and their use counts become 2. Now when we set `s1 = NULL` and `s2 = NULL`, the reference counts drop down to 1 again. The interesting thing is that the reference counts are indeed correct because `0x804c008` points to `0x804c030`, and vice versa. However, `0x804c008` and `0x804c030` have become inaccessible from the program. Thus, this is a memory leak.

As you can see, the problem has arisen because of cyclic references which make the reference counts 1 and thus falsely indicate that the structures are in use.

G.3.1 Solution Idea

This problem can only be solved using so called the class `weak_ptr` in conjunction with `shared_ptr`.

The basic idea is to break every pointer cycle by putting one `weak_ptr` in it. A `weak_ptr` is a pointer which does not increment the reference count. Further, if the object pointed to by the weak pointer is deleted, then the weak pointer becomes `NULL`. So whenever you wish to traverse a weak pointer `W`, you must first check if `*W` is not `NULL` and only then traverse. This is unlike a `shared_ptr`, which is guaranteed to point to a valid object, unless you yourself set it to point to `NULL`.

It should be acknowledged, of course, that it may be tricky to determine when a cycle is about to be formed. Thus, it is tricky to determine when a `weak_ptr` is to be used rather than a `shared_ptr`.

G.4 CONCLUDING REMARKS

Managing heap memory in C++ is an evolving field. As a novice programmer, your needs will probably be met by the classes in the standard library. If for some reason you need to go beyond that, ideas such as `shared_ptr` (and also `weak_ptr` if necessary) will likely be adequate. There is work on so called *garbage collection* strategies, but that is beyond the scope of this book.

APPENDIX H

The C++ Preprocessor

When you write a C++ program, you can designate certain words to be *short forms*, or *macro*s, which are first expanded before the compiler processes your program. Also, you can designate certain parts of your program to be compiled conditionally, i.e. only if certain conditions you specify are satisfied. All this is accomplished by the C++ preprocessor, to which this appendix gives a brief introduction.

Lines of the program beginning with a `#` are said to constitute *preprocessor directives*. There are several of these: source file inclusion, macro definitions and un-definitions, directives controlling conditional inclusion.

A preprocessor directive ends on the line it starts, unless the last character of a line is a backslash `\`, in which case the directive is deemed to continue on to the next line.

H.1 SOURCE-FILE INCLUSION

The form of this is

```
#include filename
```

You have seen this throughout the book, the directive gets replaced by the contents of the file `filename`. The filename must be in quotes or in angled braces.

H.2 MACROS

A macro definition has the following syntax.

```
#define macro-name replacement-text
```

Wherever `macro-name` appears in the rest of the file, the preprocessor will replace it with the `replacement-text`. The `replacement-text` can be anything, it need not be C++ statements or blocks.

We mentioned in Section 11.1 that the identifier `main_program` expands to `int main()`. This happens using the following macro in `simplecpp`.

```
#define main_program int main()
```

Note that it is possible to omit `replacement-text`. In such cases, `macro-name` is considered "defined", but it does not have a value. You have seen this form in Section 11.2.5.

A macro definition can also have parameters. The form is

```
#define macro-name(parameter1, parameter2, ...)  replacement-text
```

A simple example is

```
#define mymax(a,b) ((a)>(b)?(a):(b))
```

This would cause any occurrence of `mymax` in the rest of the file to be replaced by the supplied `replacement-text`, with the parameter values suitably substituted. For example, if your file contains the text `mymax(p,5)`, it would get replaced by `((p)>(5)?(p):(5))`. As you can see, this would be a way to define the `max` operation.

Note that the above macro definition contains much parenthesization. This is recommended. Suppose you did not have the parenthesization it might end up meaning something quite unexpected, depending upon the context. For example, without the parentheses, a line such as `cout«mymax(p,q);` will not even compile. A line like `z = 100+mymax(3,5);` will compile, but it will likely mean something different from what you intended.

If the replacement text contains quoted text, parameter substitution does not happen inside it. But what if you want this to happen? For this, the operator `#` can be used. An expression `#q` where `q` is a parameter will get replaced by a quoted expression containing the value of `q`. For example if you have the macro

```
#define Quote(q) #q
```

and if your program contains `Quote(3)` it will be replaced by `"3"`.

An interesting operator is `##`. It enables two symbols to be spliced together. If you have the definition

```
#define splice(p,q) p##q
```

and if your program contains `splice(a,b)` it will be replaced by `ab`.

You can "undefine" a name that you have previously defined, if you wish. Thus, following the definition of `mymax` above, you may write

```
#undef mymax
```

This would make `mymax` be considered not defined from that point onwards, and you could choose to define it differently later if you wish.

H.2.1 Predefined Macros

The macro `__LINE__` expands to the current line number in the current file. The macro `__FILE__` expands to the name of the current file. The macro `__TIME__` expands to the current time. The macro `__DATE__` expands to the current date. Note that all these names include 2 underscores at the beginning as well as at the end.

H.3 CONDITIONAL INCLUSION

The first form is

```
#ifdef macro-name
if-text
```

```
#else
else-text
#endif
```

If `macro-name` is defined, then the form expands to `if-text`. Otherwise it expands to `else-text`. You may omit `else else-text`, in which case the form expands to nothing if `macro-name` is not defined.

The form `ifndef` is exactly the reverse:

```
#ifndef macro-name
if-text
#else
else-text
#endif
```

If `macro-name` is *not* defined, then the form expands to `if-text`. Otherwise it expands to `else-text`. You may omit `else else-text`, in which case the form expands to nothing if `macro-name` is defined.

You have already seen this used in Section 11.2.5.

Finally, we also have a standard `if-then-else` like form:

```
#if condition
if-text
#else
else-text
#endif
```

In this, `condition` can be an expression that can be evaluated at compile time, including expressions involving values of preprocessor macros. Further, you can have clauses of the form

```
#elif condition
elseif-text
```

with the natural interpretation.

Index

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X