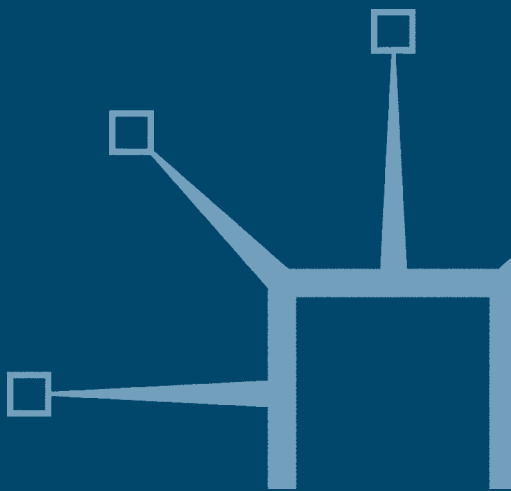
palgrave
macmillan

How to Program Using C++

Tony Jenkins



How to Program Using C++

Tony Jenkins

Illustrations by Christine Jopling

palgrave
macmillan



© Tony Jenkins 2003

All rights reserved. No reproduction, copy or transmission of this publication may be made without written permission.

No paragraph of this publication may be reproduced, copied or transmitted save with written permission or in accordance with the provisions of the Copyright, Designs and Patents Act 1988, or under the terms of any licence permitting limited copying issued by the Copyright Licensing Agency, 90 Tottenham Court Road, London W1T 4LP.

Any person who does any unauthorised act in relation to this publication may be liable to criminal prosecution and civil claims for damages.

The author has asserted his right to be identified as the author of this work in accordance with the Copyright, Designs and Patents Act 1988.

First published 2003 by
PALGRAVE MACMILLAN
Houndmills, Basingstoke, Hampshire RG21 6XS and
175 Fifth Avenue, New York, N. Y. 10010
Companies and representatives throughout the world

PALGRAVE MACMILLAN is the global academic imprint of the Palgrave Macmillan division of St. Martin's Press, LLC and of Palgrave Macmillan Ltd. Macmillan® is a registered trademark in the United States, United Kingdom and other countries. Palgrave is a registered trademark in the European Union and other countries.

ISBN 0–333–99025–0

This book is printed on paper suitable for recycling and made from fully managed and sustained forest sources.

A catalogue record for this book is available from the British Library.

10 9 8 7 6 5 4 3 2 1
12 11 10 09 08 07 06 05 04 03

Printed and bound in Great Britain by
Creative Print & Design (Wales), Ebbw Vale

This is for my god-daughter, Aimee

Using this book

Deciding what to read

This book is meant to be read by someone who is learning to program. It is not meant to be a reference. The first chapter explains what's in each of the following chapters in some detail, but briefly:

Chapter 0 – What this book is **p. 1**
... tells you in much more detail what this book is, and why it is like that.

Chapter 1 – Programming **p. 11**
... explains what programming is, and why you might want to be able to do it.

Chapter 2 – The mechanics **p. 20**
... describes the details of getting a computer to run your program.

Chapter 3 – Learning to program **p. 29**
... explains why many people find learning to program difficult, and offers encouragement to those who might be expecting to find it difficult.

Chapter 4 – Objects. The building block **p. 40**
... explains what precisely the basic component of an object-oriented computer program – an "object" – is.

Chapter 5 – A word on analysis and design **p. 52**
... puts programming into context by looking briefly at the processes of analysing a problem and designing a solution.

Chapter 6 – Programming (don't panic!) **p. 63**
... shows why programming requires a structured, controlled approach, and why good programming style is so important.

Chapter 7 – The basics **p. 74**
... introduces the first C++ in the book – values, variables and assignments.

Chapter 8 – Input and output **p. 92**
... describes how to display values to a user and how to accept input values from the user.

Chapter 9 – A word on testing **p. 106**
... breaks off briefly to show why it is so important that all programs are tested thoroughly and methodically, and to present some ways of testing simple programs.

Chapter 10 – Defining C++ objects **p. 118**
... begins the introduction of C++ objects by describing how to define an object type.

Chapter 11 – Using C++ objects **p. 132**
... continues boldly on by showing how to use a C++ object in a program.

Chapter 12 – Get your hands off my data! **p. 146**
... describes two basic functions that are carried out on objects – setting their interesting values and finding out what these values are.

Chapter 13 – Making things happen. Sometimes **p. 161**
... returns to the basics of C++ and describes how to make a program behave in different ways in different situations.

Chapter 14 – Making things happen. Again and again **p. 184**
... extends the previous chapter to show how to make a program repeat an operation over and over again.

Chapter 15 – More methods **p. 201**
... shows how to use the techniques from the preceding two chapters in implementations of object types.

Chapter 16 – Making things happen. In bite-size chunks **p. 221**
... describes how C++ programs can be broken down into small easy-to-handle parts to make the development process quicker and easier.

Chapter 17 – Collections of things **p. 244**
... concludes the description of features of C++ by showing how programs can be written to handle large collections of data rather than just single values.

Chapter 18 – A case study **p. 269**
... ties the chapters together by describing and illustrating the process of developing a program from specification to final implementation.

Chapter 19 – More on testing **p. 296**
... reminds us that all programs should be tested and that more complicated programs require more thorough testing still.

Chapter 20 – Onward! **p. 316**
... rounds off the book with brief descriptions of a few of the more advanced C++ features.

If you're approaching programming as a complete novice you should aim to work through these chapters in this order. Don't be tempted to skip straight to the chapters with C++ in! It's extremely important that you understand what you're trying to achieve and the best ways of achieving that before you go anywhere near any programs.

If you've already programmed in some other language (particularly something like Pascal or C) and just want a flavour of C++, you're probably safe to skip on to Chapter 6 or 7. It would still be a good plan to skim through the earlier chapters, though.

At the end of the book you'll find a quick C++ Reference, a Glossary and an Index. This is where to look when you realise that you need that little bit of information but can't remember where it was.

Understanding what you read

There are some conventions that you need to keep in mind as you read.

In the text, anything in `this font` is a C++ statement or name. Anything in this font is not.

All programs also appear in `this font`. Like this:

```
// hello.cc
//
// The Traditional First C++ Program
//
// Display "hello, world" on the screen.
```

```
//
// The Program before the Book.
//
// Author:    AMJ
// Date:      20th October 2001
// Platform:  g++, Linux

#include <iostream>

using namespace std;

int main ()
{

  cout << "hello, world" << endl;

  return 0;

}
```

Anything not in `that font` is not a program. Fragments of programs also appear in `this font`:

```
cout << "Enter your name: ";
cin >> name;
```

Anything in `this font` is correct C++.

Where a user is entering a value, the user's typing is shown in bold:

```
Enter your name: Tony
```

Sometimes it is necessary to show just the general format of the C++ statement. This appears like this:

format of a C++ statement

For example:

<type> <identifier>;

Anything that appears between < and > in these examples is a description of what is required in the program. Examples presented like this are not valid C++!

Finally, sometimes there are things that need to be laid out in a way that looks like a program, but isn't a program. This appears in the same font as a program, but in *italics*, like this:

```
IF the value of "sold to" is blank THEN
  Set the value of "sold to" to the Farmer object
OTHERWISE
  Display an error message - the sheep is already sold
END IF
```

This is not valid C++.

There are many definitions in the book. Words that are defined in the Glossary at the end appear *like this* in the text.

A note on programming style

You will learn that programming style is an important element of writing a program. Style refers to the way in which a programmer lays out a program,

what names are chosen for various things in the program, and what else might be added. Programming style is a very individual thing, and develops in any programmer over many years; it's very much like handwriting.

The programs and examples in this book were written by me, and so are in my own preferred style. I am not by any means claiming that my style is a particularly good one! It has evolved over many years and has passed through many languages. Looking through these programs now, I suspect that the style is a little terse in places; this will be something to do with my heritage of writing C. If you find it so, then I apologise.

Develop your own style as you learn to program. Copy mine for the moment if you want to, but if you find things that you don't like, don't follow them. Just be consistent. A world full of C++ programmers who all lay out their programs in the way that I do would be a very worrying place....

To the Student

Welcome to the book. I hope you like it.

If you're a student just starting out on your first programming course, this book is for you. I've met a lot of people just like you over the last few years and I hope I've now got a pretty good idea of what you're feeling about your programming course at the moment. This book contains what you'll need to know as you go through your course. I hope you'll enjoy reading it and come to enjoy programming.

I want to get something absolutely clear before we go any further: programming *is* enjoyable. It's a creative pastime, and has been called by some a "craft". Writing a program is the process of creating something from nothing – the process of creating something that solves a real problem and hopefully makes the world a better place. A complete and correct program can be a source of great satisfaction to a programmer. Even the appearance of the lines of a program laid out on a sheet of paper can be a thing of beauty, almost like a poem.

But enough of these fine words. I would be lying to you if I didn't admit that many people do not especially enjoy learning to program. Many people do find it difficult, but just as many take to it quickly and easily; I suppose that you'll be finding out which one you are in the next few weeks. Whichever turns out to be you, just keep in mind that anyone can get there eventually; there's nothing special about people that can write computer programs.

This book is not like many of the other books on programming that you can see on shelves in the bookshops or libraries. For a start, there's no chapter on the history of computers, gloriously illustrated with highly amusing photos and hairstyles from the 1950s. No – I'm assuming that if you want to look at that sort of stuff you know where to find it, and you'll go and seek it out. No. This is a book about *programming*.

You notice that I say *programming* and not *C++*? That's important. What we are about here is learning to program, and the programming language that we have chosen to use is something called C++. The skills and techniques that you'll find in this book, and which you'll learn, can be applied to many, if not most, other programming languages. A mechanic does not learn to repair just one kind of car, and a chef does not learn to cook just one kind of pie.

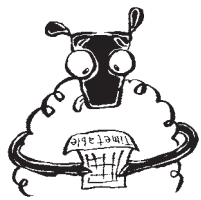
Just reading this book won't turn you overnight into a programmer. Reading this book will help turn you into a programmer, but you're going to have to do other things. You're going to have to write your own programs. You're going to have to practise. There are plenty of my programs in this book, and the first stage for you is to look at these and understand them. Then you're going to have to take the leap of starting to write your own programs. This transition – from understanding a program that someone else has written, to writing your own programs – is difficult, and let no one tell you otherwise. When you've managed it you'll have achieved something that should make you very proud.

The chapters that you're going to read are quite short. This is quite deliberate. With any luck you'll be able to find time to read them between classes or on the bus home (or wherever). As you read, doodle on the book, or make

notes. Read *actively* and think about what you're reading. Learning is about thinking and reflecting, not just about reading.

Right. Sermon over. There's one last practical thing. You'll find out soon that I'm going to assume that you've been given a copy of something that I'll call your *Local guide*. This should explain how the C++ system that you're going to use works, and should fill you in on any other little local details. If you got that (and there's no need to look inside it just yet), all that remains to be said is ...

... let's go and do some programming!



To the Teacher

Welcome to a book about learning to program.

Before we go any further, you need to be absolutely clear about what it is that you are holding in your hand. This is possibly a book with the name of a programming language in the title that is unlike any book with the name of a programming language in the title that you have encountered before. And I expect that you have encountered many.

A big claim, that. But this is not a book about C++. This is not a book that seeks to explain all the minute details of the C++ language. This book contains no UML and none of whatever the flavour of the month is at the moment in systems development. This is not a book that an experienced programmer, working in industry, would use as a reference as they worked on some commercial project. There are lots of books like that, and lots of books written for experienced programmers, and this is not one of them.

This book came about like this. I've been to a few conferences on teaching computing, and I've given a few presentations describing some of my ideas on what's wrong and right with the way we teach programming. I've come to the conclusion that there's rather more that's wrong than right. A publisher's rep came up to me at one of these happy events and started to pester me to write my own C++ book. I declined, since there were already far too many C++ books about and I saw no need to add to this needlessly large pile of paper.

The problem that then emerged was that this was a persistent publisher's rep. I kept finding that she kept popping up in my office. I will admit to having been bought a beer, but despite advice from other authors, I always seemed to miss the free lunch. Eventually, during ITiCSE 2001 at Canterbury, I cracked and agreed to write something. But only on my terms. I was not going to write another totally unnecessary book about C++.

That is why this is a book about learning to program. Specifically, this is a book that is intended to support a student following an introductory programming course in further or higher education. There is sufficient C++ in this book to be included in such a course; there are also some pointers in the final chapter that would be of interest in the more ambitious courses[1]. My hope is that after reading this book, and after following your course, a student would be able to write some reasonably complex C++ programs and make sensible use of one of the many other C++ books that are available.

Now let me explain why this book is like this. I have taught programming for many years in what is probably one of the most respected university computing departments in the UK. Every year I have some successes, and every year there are failures. I see students struggle with this topic; they are struggling with something that lies at the very heart of our discipline. I often see students suffer as they attempt to come to terms with programming; often I have seen them drop out of their degree simply to avoid more programming.

---

1 But those teaching more ambitious courses would do well to ponder whether it is better for a student to understand a little C++ or to be totally baffled by a lot of C++.

I have certainly seen them carefully choosing course options in future years to avoid anything that resembles programming[2]. Your students might be different, but somehow I doubt it. This sad state of affairs just cannot be right.

One aspect of this problem (or at least one issue that contributes to the problem) is the nature of the programming textbooks available. These are often weighty tomes indeed, and many run to well over 1000 pages. You know the ones I mean. Most contain far more than could ever be learned effectively in a single course that is, after all, only one part of what a student is expected to study during the year. These books (there are a few honourable exceptions, of course) simply do not meet the needs of our students.

This book works like this. It not my intention, or my place, to try to replace your lectures. My job is to support your lectures by providing something that your students will actually read, hopefully before your lecture. My job is to explain to them what's coming up, why it's important, and why it's useful. Each chapter should occupy about a week in a 20 week course; for a student this week should probably include a couple of lectures, some supervised practical time, and opportunities for plenty of practice. There are some exercises at the end of each chapter; please add in some of your own to fit your own local system or needs.

You might think that sometimes my explanations are a little simplistic. Sometimes I admit that I am, in Civil Servant terms, "economical with the truth". This approach is essential with a language as complex as the language that C++ has now become; there are so many little details that can tend to get in the way of the real business of the day, which is to learn to program. Sometimes I've added a more complete explanation as a footnote; you might well want to go into more detail, particularly with your more experienced or advanced students. It's up to you.

I'll give you one specific example. I've included `using namespace std;` in all the programs in the book, and in all the header files. This would make some experienced C++ programmers cringe, especially the use in the header files. That's fine, and the experienced C++ programmers are quite right. But if `std` is the only namespace you ever use, it's also fine to include it everywhere. Once your students understand (or need to understand) precisely what a namespace is, they can understand why including this line everywhere is not the best idea.

Some C++ books come with a handy C++ compiler, kindly provided free-of-charge by one of the larger software companies. This book doesn't. I prefer software that is free and comes with no strings attached. All the programs in this book have been written and tested using version 3.0 of the GNU *g++* compiler running on a Linux platform. I believe that all the programs work (except where stated otherwise), and should work unchanged with other C++ compilers, even commercial ones and those written by one of the larger software companies.

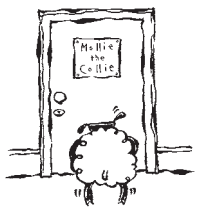
There's one last thing I need you to do before we start. I don't know what C++ system you're planning to use. I know nothing about your editor-of-choice, and I don't even know what operating system you're using. I've ignored all these issues, since I want this book to be useful to everyone.

---

2 At Leeds there is a second year course on Linear Programming. They avoid that too. Just in case.

Obviously, though, there are some things your students need to know. As you read through the book you'll see that I've told them about something called the *Local guide*[3]. I need you to put this together for me. Here you can describe how your system works, you can set down coding or layout standards if you want to, and you can pass on any other local wisdom. You could probably combine it with some existing set of notes. I hope that's not too much work for you. The exercises at the end of Chapter 2 include most of the things that I need you to make sure that your students know.

Finally, I hope you agree with my reasons for writing this book. If you do, we're ready to go and teach some programming!



3 If you were to suspect that I've "borrowed" this idea from Leslie Lamport's LaTeX book, you'd be quite correct.

The web site

This book is accompanied by a web site. The address is:

http://www.comp.leeds.ac.uk/tony/book/c++/

Details of the book and access to the web site can also be found at:

http://www.palgrave.com/science/computing/

On the web site you can find:

- All the code that you'll need to complete the exercises.
- An interactive C++ reference using the examples from the book.
- All the solutions to the exercises.
- Additional exercises for every chapter.
- All the programs from the book, so that you can download them, try them out, and adapt them.
- Links to free C++ compilers, editors and other development tools.

There are also forms to submit comments and useful links, and much more. If you require a hard copy of the solutions to the exercises please contact the publishers.

About the author

Tony Jenkins is a Senior Teaching Fellow in the School of Computing at the University of Leeds. His hobby is teaching introductory programming. He is lucky that his hobby is also his job. He has given many presentations and written many papers about the ways in which programming is taught.

Tony gained his BSc from the University of Leeds in Data Processing (back when computers were *real* computers) in 1988. Five years spent writing programs in what many call the "real world" convinced him that fun was more important than money, and he returned to the University of Leeds in 1993. He has been teaching since then, and teaching introductory programming since 1995. In 2002 the University of Kent at Canterbury saw fit to award Tony an MSc for research into the experience and motivation of students learning to program.

When not at work, Tony can generally be found with Dave and Wallace in the Grove or the Eldon, probably after two hours on the terraces at Headingley observing the antics of the Tykes or the Rhinos. He has been known to drink beer.

Tony approves of cats, but is allergic to them. He thinks that sheep are all right, but he has never been closely acquainted with one.

Anyone wanting to contact Tony about this book (or just to have a chat) can send email to *tony@comp.leeds.ac.uk*. There are rumours of some sort of web page at *http://www.comp.leeds.ac.uk/tony/*, but *http://www.leeds-camra.com/* is probably a better bet.

Acknowledgments

There are many people (and, happily, no sheep) that I need to thank for their help in the production of this book.

First thanks go to Mark Wiggins. I suspect that we learned to program together. All those hours hunched over the ZX81 and Spectrum have come to this – thanks. And belated thanks for lending me the Speccy for the O level coursework.

Various people at Palgrave Macmillan have egged me on for what seems like a terribly long time. Tracey Alcock believed I could and should do this and nagged me until I said I would. Esther Thackeray and Becky Mashayekh have kept me going since. Everyone seems to have taken the idea of a C++ book with so many sheep in it remarkably well.

Fintan Culwin also seemed to think I could do this and provided many useful comments on the early drafts. Somewhere in the Shepherd Neame Brewery Xristine Faulkner assured me that Fintan wasn't having me on.

Nick Efford, John Davy, Vania Dimitrova and Karim Djemame all read through the final drafts of the chapters, spotted mistakes and provided many helpful suggestions and comments. Chris Gillespie used early drafts to support a C++ class and provided useful feedback. Lee Curtis passed Chris's class and wrote me some useful comments. Any errors that remain are, I suppose, my fault.

Ian Kingston did a fine job of turning my collection of Word files and images into a book.

Ian Fletcher and Richard McLeod developed the web site. Thanks to Sarah Fores for fixing this up.

Christine Jopling provided the illustrations and never seemed to mind too much when I asked her for just one more. These illustrations are all © Christine Jopling 2003. Dave Ansley and Andrew Amies-King sorted me out with the electronic versions of the illustrations.

Graham Hardman provided the Java code in the Further Reading section. Margaret Jenkins provided the chocolate cake recipe. Steve King gave me the CDs.

The Technical Support crew at the School of Computing at the University of Leeds provided the computer systems on which this book was created. Special respect goes to *cslin101* and *cspcx27*, which have both taken something of a hammering. Special thanks to Jim Jackson for finally pointing me in the direction of a sensible way of getting C++ programs from a Word file onto a Linux machine and back again.

I must confess that Flossy is not my idea. Flossy started out life as Cheryl. Cheryl was a sheep who rode a motorbike in a game written by Phil Mellor and Clare Viner for my Computer Graphics course. Flossy, or something that looked very like Flossy (or Cheryl), also appeared in programs developed by Phil for one of my Introductory Programming courses. I'm pretty sure that Christine Jopling provided the name. Dave Ansley named the cow and agreed to change the name after I'd explained a few things about cows (who is called Clare, even if the name doesn't seem to have made the text).

Phil Mellor is also responsible for the idea of C++ programs to track creatures moving around a grid. This is what happens when Phil devises introductory C++ exercises. In Phil's case the creatures were (predictably enough, if you know Phil) orang-utans, called Bob and Trev, who moved around a jungle. While Flossy and her friends sometimes meet a big hole in their field at (5, 3) (and while Clare often finds the milking shed in much the same place), Bob and Trev used to find mangosteems in the same spot; this is a foodstuff that is, according to Phil, much prized by orang-utans. This information can be confirmed at the excellent `http://www.orangutans.com/`.

Less specific thanks are due to Sylvia Alexander and all at LTSN-ICS for providing a forum where I could discuss ideas with like-minded folk. Thanks also to said like-minded folk, David Barnes and Janet Carter from Kent, Pete Chalk from what I believe is now called London Metropolitan, Fintan Culwin from South Bank, Hugh Davis and others from Southampton, John English from Brighton, Lachlan MacKinnon from Heriot-Watt, Glenn Rowe from Dundee, and many more.

Even less specific thanks is due to the rest of the teaching team in the School of Computing. Respect is due to Liz "Cruella" Minton, Andrew Amies-King, Chris Gillespie and Ollie Clark for just being around for a chat when things get tough.

Finally, I must thank all the students who have so patiently endured my attempts to teach them a little bit of programming. No permanent harm done, I hope.

Thanks.

Tony Jenkins
Leeds, 2002

This page intentionally left blank


Chapter 0
What this book is
ADA COLLIE
CHARLES COLLIE
Alah Collie
Grace Collie
Chop
SHEEP

Good Evening. Glad you could make it. Pull up a chair. Let me put some more logs on the fire. Move the papers off the chair. Help yourself to a drink from the cabinet; there's some orange juice at the back. I'm afraid I've run out of toast. Would you like a chocolate digestive?

I see you've got a copy of the new book *How to program using C++*. A wise choice if I may say so. While you have your drink and digestive, just let me explain what all this is about. There are some things that you're going to need to know before we start.

This is Chapter 0 and that is your first lesson. Most books have Chapter 1 as the first, but this one has Chapter 0. I'm afraid that computers are like that. You're going to have to get out of the habit of starting to count from 1. Computers have a nasty habit of starting to count from 0, and if we're going to write programs to make a computer do useful things for us we're going to have to get into that habit too.

Before we go any further allow me to explain what this book *is* and more importantly what this book is *not*.

What this book is not

This is not a book about C++. There are plenty of books about C++ available and it would have been very foolish of me to spend my precious leisure time trying to add to their number. This is not a C++ reference book, although there is a handy C++ reference in the back. It is not a book to read if you are already confident about programming and can program well in some other language. No – this book is for people who can't program. It is for people who don't have much idea of what the whole thing is about. It is perhaps even a book for people who are starting a course in programming and are just a little bit worried about it.

What this book is

This is a book about learning to program. There are not many books about learning to program, even if some books about programming languages claim to be about learning to program. These claims are normally wrong. This is a book about learning to program, and the language we will use to write programs as we learn is C++. This means that you will also learn some C++, but don't confuse the two things. If you can write programs in C++ you can quickly learn to write programs in other languages.

Computer programs are written in computer programming languages. Deep down most programming languages are basically the same. They use the same concepts and ideas and much the same constructions. Sometimes the way of achieving something is the same in several languages. If you can learn to program in one language you can normally pick up another language without too much bother. I originally learned to program in a language called BASIC. Then I learned another called Pascal and then something else called C. Finally I arrived at C++, the language in this book. It's learning the first language that's the difficult bit; it's much, much easier after that!

This is a book about learning to program. It is not a book about a particular programming language. This means that this book is rather unusual. There is, not surprisingly, a lot in this book about learning to program, but not very much just about C++. When you've read this book and when you've done all the exercises you'll be able to get a different book about C++ and learn

about all the extra fiddly little details that I've missed off as and when you need them. The last chapter will point you in the right direction. My job in this book is to put you in a position where you can use one of the other books if you want to learn more C++ or another language.

You are learning to program. You are not learning C++. Promise me that you'll remember that.

Who this book is for

This book is mainly for students following a first programming course in C++ as part of a further or higher education course. This will probably be in the first year of the course. It is especially for those students who have never done any programming before. It doesn't matter what your main course is. Whether it's computing or something totally different, if you've not done any programming before this is the book for you.

It really doesn't matter at all if you've done no programming before. In fact, about half of the students I meet every year before their programming course starts have never done any. If you've done a bit before you might want to let your attention wander in a few places. I'll trust you to spot when these are.

Of course, other readers are welcome to join us. If you just want to know something about programming or object-oriented programming in particular you're in the right place. The more the merrier, that's what I say.

Why this book is like this

I've been teaching people to program for a very long time. Every year I teach many students and every year almost all of them succeed. When they find things difficult they often read their book. Sadly their book has almost always been written for people who can already program or for people who find the whole thing very easy or by people who don't understand what it's like to learn to program. Most of my students do not find the whole thing very easy, and I hope I understand why. This book is for them.

Often my students tell me that they don't understand something or that they're stuck with some programming topic. They tell me that they've been away and read their book, but that it didn't seem to help. This doesn't surprise me. This is the book I wish they'd had.

Without further ado let me explain what is in this book.

What is in this book

After this introduction every chapter[1] in this book has five sections. The sections work like this:

In brief This section introduces you briefly to the topic of the chapter. It tells you what the chapter is all about and tells you what you should understand after reading it. Think of it as an enticing appetiser. Our friends who have done some programming before might want to read this to help them to decide whether they need to read more.

---

1 All right, I admit it. I mean "almost every".

The idea This is the main part of the chapter. It explains the new idea that is being introduced in some detail and illustrates it with sample programs. This is the section you should pay the most attention to. It is the hearty main course.

Examples Most chapters also have some examples. These use the new ideas from the chapter and apply them to a new problem. I'll show you the correct solutions and working programs and explain why they are correct and why they work. This is the dessert and is something to look forward to.

Exercises There are always exercises or tasks for you to try yourself. They are not optional! You should work through these on your own, and don't start looking at the answers until you have. You should linger a while over this. Think of it as the cheese board.

Summary Finally, you'll find a brief summary of the chapter. This explains what you should have learned and what you should now be able to do or what you should understand. This is what you should be able to do before going on to the next chapter. The coffee and mints.

You should try to read each chapter in one sitting; they're not very long. Hopefully you'll be able to read one on the bus on the way home or something. Then try the exercises later on. Don't be tempted to read the book while sitting at a computer.

This book uses an approach to object-oriented programming called "objects first". We'll start by looking at what "objects" are and how we might recognise one in the real world. Then we'll move on to see how we can implement programs using objects in C++. Some C++ books don't deal with objects first; many years of experience have convinced me (and others) that they're wrong.

The chapters

The chapters are short, so there are quite a few of them. Here's what's in them. Read this now and you'll see where we're going.

Chapter 0 What this book is You are here! I want you to understand why this book is like this. You need to know what's going to happen in the rest of the book. Most importantly, you need to realise that learning to program and learning C++ are not the same thing.

Chapter 1 Programming The book starts with a bit of background and a quick history lesson. Before you start to learn to program it's important that you understand what a computer program really is. You'll learn that programs are all around us in all sorts of surprising places.

Programming has developed over many years. Over this time there have been many different approaches and many different languages. C++ is the programming language you'll use. In order that you appreciate why it is as it is we'll also have a brief history of programming and of C++ itself.

Chapter 2
The mechanics

You are going to learn to write programs. You are also going to have to learn how to make your computer execute your programs to make them actually do something. You are going to have to know how to find the results that your programs produce.

What precisely you have to do will depend a lot on the computer system that you're going to use; you might be using Microsoft Windows, Unix or even something else. I don't really mind.

This chapter gives you the general idea of how to create and execute a program and will tell you what you need to find out and where to find it. It explains what goes on behind the scenes when you prepare to run one of your programs.

Chapter 3
Learning to program

Learning to program is difficult, but it is not impossible. It will help you as you learn if you understand this and if you understand why it is often found so difficult by so many people. Finding it difficult is quite normal!

This chapter explains why it is so difficult and describes how to think about the best way to go about learning. If you are going to succeed, you will probably need to think about the ways in which you learn best. This chapter tells you how and why.

Chapter 4
Objects.
The building block

C++ is an example of something called an *object-oriented* (and not object-*orientated* – let's get that right from the start!) programming language. This chapter explains what this means by explaining precisely what an object is and how you might identify one.

Objects have special properties called *attributes* and *methods*. This chapter also explains what these are and how you can choose the correct set for the types of object in your programs.

Chapter 5
A word on analysis and design

Before anyone can write a program someone must decide precisely what the program should do. This involves analysis of the problem that the program is going to address and design of the solution.

This is really only a book about programming but you will also need to be able to do some basic analysis and design. You will need, for example, to analyse a problem you have been given to solve and then design a solution. This chapter explains what you need to know to do just that.

Chapter 6
Programming (don't panic!)

Programming is a tricky business. The title of this chapter gives the best advice of all for new programmers – Don't Panic! Programming is a structured activity that requires and even demands a structured, methodical approach.

Many new programmers do indeed panic when faced with a new problem. They make mistakes that an experienced programmer would never make. They make mistakes

and work themselves into a hopeless state from which they can never recover.

This chapter describes the process of writing a program and some of the common pitfalls and mistakes that new programmers make. Hopefully after reading it you won't make them – or at least you'll realise when you do make them so you'll only make them once.

Chapter 7 The basics

This is a third of the way through the book and this is the first chapter that includes any C++! Don't be tempted to skip straight to it though; the preceding chapters are there for a purpose and contain essential background that you will need to have read and understood.

This chapter introduces the idea of a *variable*, the most basic component of a program. It shows how variables are used to store and manipulate values held in a program. After reading this chapter you should be able to write your first C++ program.

Chapter 8 Input and output

To write properly useful C++ programs you need to be able to get values from a user. You need a user to input these values, usually from the computer's keyboard. Your program should then output the final results that it produces, usually on the computer's display. This chapter explains the C++ you need to do this. After reading it, you'll be able to write some useful C++ programs.

Chapter 9 A word on testing

For a program to be truly useful we need to have confidence in the results that it produces. We need to believe that they're accurate and correct. This means that we have to test all our programs very thoroughly. Testing is another structured process that requires a plan.

This chapter explains how to build up a series of *test cases* into a *test plan*. This plan will allow you to test your programs so that you can be confident that they work. If you are confident in the results produced by your program your users should be too.

After reading this chapter you should also understand why it is never possible to be completely sure that any program works in every possible case!

Chapter 10 Using C++ objects

Armed with an understanding of some basic C++ and an appreciation of what an *object* is, we can move on to create some simple programs that use objects.

Objects are implemented as instances of something called *classes* in C++. This chapter describes some classes and shows you how to write programs that use them.

Chapter 11 Defining C++ objects

Now that you are able to use objects in C++ it would obviously be a good thing if you could define your own. This chapter explains how to do just that and describes how to create the necessary program files – the *header file* to contain the description and the *implementation file* to contain the details.

At the end of this chapter you should be able to write simple C++ classes and programs that make use of them. You should be able to identify the classes in a problem area and design and write a program using them to solve the problem.

Chapter 12 Get your hands off my data!

C++ classes have a public face and a private face. The public part, called the *interface*, is available to all programs that use the class. The private part, on the other hand, is available only to the class itself. Classes are not unlike people.

This is a very powerful mechanism that allows for *data hiding*. It is one of the key ideas that make C++ programs portable between different computer systems and one of the properties of C++ that makes it one of today's most popular programming languages.

A well-written C++ program written on one computer should be able to run unaltered on another computer system. It should be obvious why this is a good thing.

After reading this chapter you should understand the difference between the public and private parts of a C++ class. You should be able to design classes with suitable public and private parts and you should understand why this means that C++ programs can be portable and reusable.

Chapter 13 Making things happen. Sometimes.

This chapter moves on to deal with the rest of the basics of C++. It describes the two *conditional statements*. Sometimes there are parts of a program that we want to execute only if some condition is true; a word processor program should print a document only if the print button is pressed, for example.

This chapter explains what a *condition* is and how it can be either true or false. It explains how to combine single conditions into more complex expressions using Boolean logic. Finally it explains how you can use such expressions to control how your programs behave in certain situations.

After reading this chapter you should be able to write more complex C++ programs that can carry out different tasks depending on the user's input and on other values. Your programs will be able to deal sensibly with unexpected input values and you will be able to implement simple menu-based systems.

Chapter 14 Making things happen. Again and again.

Sometimes parts of a program must be executed many times. This might be for some number of times that is always known (*determinate*) or for an unknown number of times until some event happens (*indeterminate*).

This is achieved in C++ with *loops*. This chapter describes the different kinds of loops available in C++ and explains how to use them.

At this point you should be able to write many useful C++ programs. You will have learned most of the basic C++ you will need and will hopefully have had a good amount of practice. You will be a C++ programmer!

Chapter 15 More methods

The new C++ in the previous two chapters will enable us to write some more complex methods in our classes.

This chapter provides some examples of doing that and also fills in some of the final details in how methods are written and used.

Chapter 16 Making things happen. In bite-size chunks

By now the C++ programs that you will be writing will be becoming quite long. Complex programs are often difficult to understand and maintain; to keep this under control they are never written as a single unit. Instead, as part of the design process, they are broken down into smaller parts called *functions*. These functions are then later assembled into complete programs.

This chapter explains how to use functions and how to design and write a program that makes use of them.

Chapter 17 Collections of things

Often programs must process collections of data. The final part of C++ that you need is the ability to store such collections. You might want to write a program to store the details of all the books on C++ in a library, for example. This program might need to search for a particular book or display a list of all the books.

This chapter explains two ways to do this in C++ using *vectors* and *arrays*. These two ideas will allow you to design and write complex and useful C++ programs. You will indeed be a C++ programmer!

Chapter 18 A case study

As you get near to the end of the book you have learned many things. This chapter ties them all together with one final big example.

This chapter shows you a problem and explains how to analyse it, design a solution, and then write a C++ program to solve the problem. This is exactly what C++ programmers do, and now you should be able to do it too.

Chapter 19 More on testing

With bigger programs you need bigger and better testing if you are to be sure that the programs work. This chapter explains how to test more complicated programs and how to test C++ classes so that they can be used elsewhere in other programs and by other programmers.

Chapter 20 Onward!

And this is the end of the book! You won't have seen all the C++ that there is (very few people have) but you'll have seen, used and practised a fair bit. This chapter introduces you to some of the more important and useful aspects of C++ that you have not seen and explains how to find out more about them.

C++ reference and examples As you write your programs you will often want quick access to a short summary and some simple examples of how the basic C++ commands and ideas work. You will want to check the details of the syntax of some command or you will need to be reminded of some detail.

This is what you need!

Further reading After reading this book you will probably want to move on to more programming. This section points the way to where you might start and gives you some references to follow up both on the World-Wide Web and in books if something has particularly interested you.

Some languages that you might want to move on to sooner rather than later are Java, Python and C#. If this is what you want to do, you'll find some handy pointers at the end of this chapter.

The book is rounded off with a handy glossary and index.

The characters

All good books on programming have a cast of lovable characters; it has to be said that in some books they're the best bit. Not wanting this book to be an exception to this fine tradition[2], I too have assembled a suitable cast of lovable characters to guide you on your way through the book.

Here they are.



Flossy The star of the show. Flossy's family business has long been wool production, but she is now deeply concerned at the current dip in the market for wool and is worried about what this might mean for her future. She has heard about mutton and kebabs and does not much like the sound of either. She has decided to branch out from her wool production business into computer programming.

Flossy is a sheep that is going to learn C++.



Freda Freda is Flossy's best friend. Freda is also worried about the wool business. Freda is with Flossy.

2 There are many fine traditions that should be observed by all books on programming. I hope that I haven't missed any; I'll be pointing them out as we go along.



Mollie

Mollie is a typical working Border Collie. Like most such dogs found on farms today, Mollie is an experienced C++ programmer. She has agreed to teach Flossy and her friends.



Peter

The wool market is not an especial worry for Peter. However, he has heard of something called "bacon" and has also decided to join Flossy's programming classes. Just in case.



Farmer Strawstrap

Farmer Strawstrap owns the farm on which Flossy and her friends live.

He knows little about C++, but is not afraid of using the skills of his animals to make the management of his farm easier.

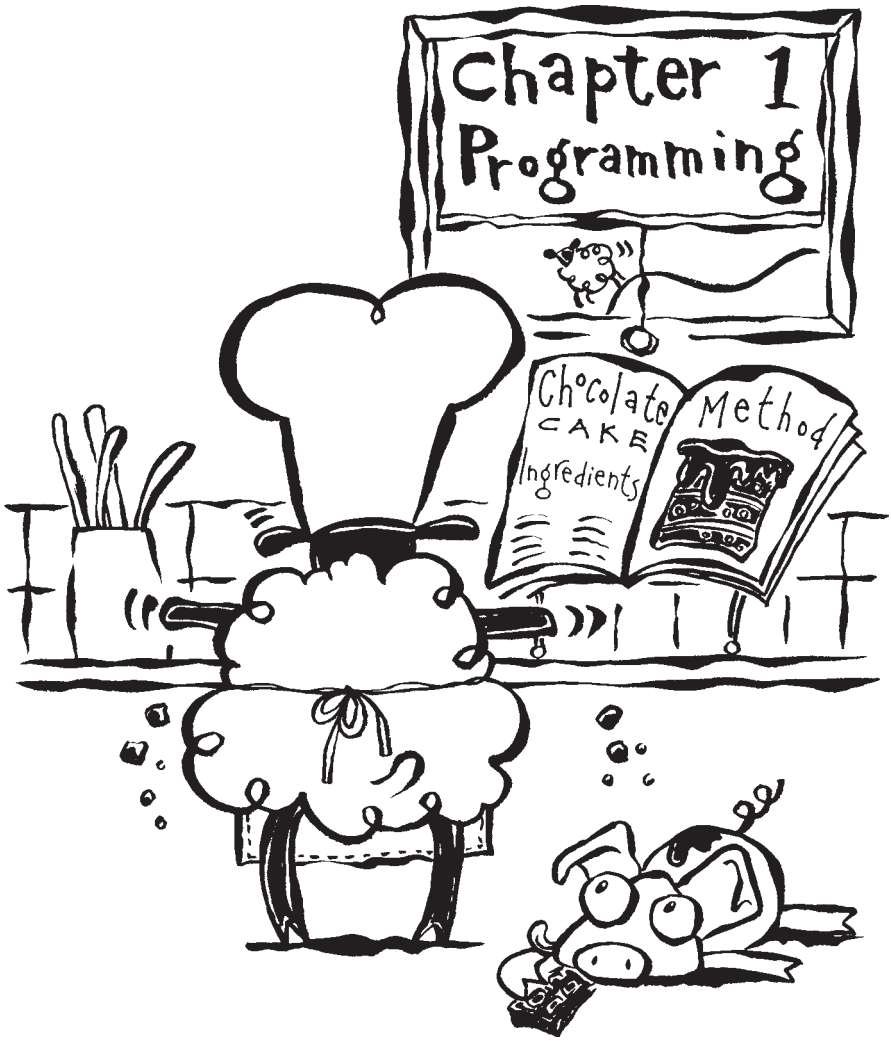
The end of the start

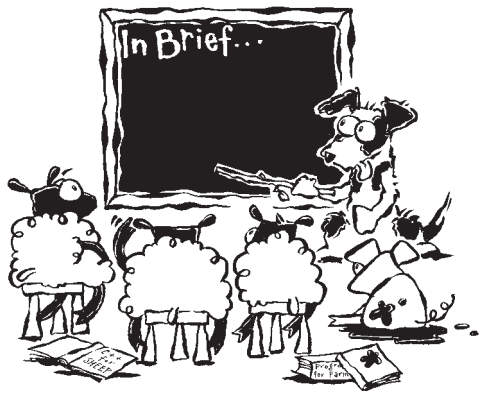
This is the end of the start of the book. Hopefully you now know what's coming up and why it's coming up in that way. It's time to read on.

Put the papers back on the chair and I'll put the fire out. Leave the empty mug on the table. Mind the crumbs.

Now, it's on with the show. Remember that we're in this together, and that I'm right behind you!

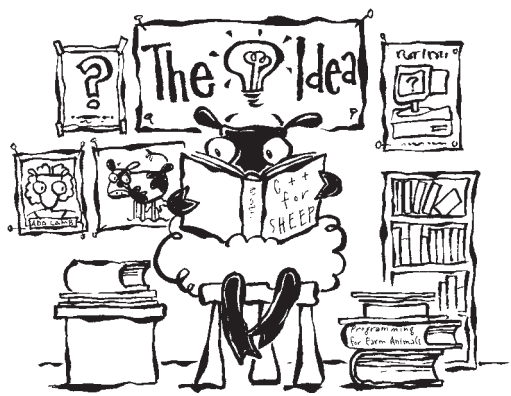



Chapter 1
Programming
Chocolate
CAKE
Ingredients
Method



Computer programs are all around us in the modern world. They are sometimes hidden away in the most unexpected and unlikely places. All of these computer programs have at some time been written by a human programmer using the processes and skills that you are about to learn. This programmer has gone through the stages that you are going to learn about in this book and has probably used a programming language not unlike C++. At some point in the past this programmer learned to write programs. Before you start learning to write programs it is essential that you know what a program is.

After reading this chapter you should understand what a computer program is, and you should understand what programming is all about. You should be able to identify the various programs that run on the computers you use and you should be able to make a reasonable guess at how they were written. You should understand something about C++ and why it is a popular programming language. You should know about some of the people who have contributed to the development of programming and to C++ in particular.



Computer programs are everywhere. I am using a computer program to write these words. These words will be processed by many more computer programs before you read them; an electronic mail program will be used to send them to an editor, a backup and compression program will be used to keep a safe copy, and the typesetter will use yet more programs. It is entirely

likely that a computer program was involved when you bought this book: there would have been one in the till when you paid; you may have used a computer-based catalogue to find the book you wanted; and you may have paid with a credit card. All these things rely on computer programs, and all these programs have been developed by human computer programmers.

While I type these words, my computer is running many other programs. The operating system itself (Microsoft Windows as it happens) is a program. I am using yet another program to choose which track to play on a CD; this program is using another buried deep inside my CD player to translate the data stored on the CD into sound. The CD itself was probably recorded on a machine that used many computer programs. Many of the instruments played to make the recording also contained computer programs.

Computer programs can be found in devices ranging from mobile telephones through dishwashers to sports cars and fighter aircraft. All of these programs have one thing in common: a human programmer wrote them and went through much the same processes for each one.

Computers are pretty useless things without programs. On its own a computer is little more than a collection of wires, plastic and tiny bits of silicon. It would make a handy doorstop, and little more. It is only when a program is added to this collection of components that a computer becomes a useful tool. The development of computer programs is a fundamental part of the development of computing.

A program

Here is a simple program. It is not the sort of program that can be used on a computer, but it *is* a program. It is, in fact, a recipe for chocolate cake[1].

Ingredients

4 oz soft margarine
4 oz caster sugar
2 size 3 eggs
3 oz self-raising flour
1 oz cocoa powder

Method

Pre-heat the oven to gas mark 4, 325 °F, 160 °C.
Beat the margarine and sugar together until the mixture is almost white.
Beat the eggs one at a time into the mixture.
Sift the flour together with the cocoa powder and carefully fold it into the mixture.
Divide the mixture into two 6 inch round tins.
Bake on middle shelf for 20 minutes.
If desired, cover with melted chocolate.

(Source: Home Recipes with Be-Ro Self Raising Flour*, 37th Edition, Rank Hovis MacDougall, 1982)*

Although it may appear to have little to do with computer programming, this recipe shares many features with computer programs:

---

1 You can try it, if you like. It's really rather good.

- It follows a format dictated by convention – all recipes look like this, with the ingredients listed first and then the method described step-by-step.
- It is written in a specialised form of language.
- This language has a vocabulary that the user must understand – beat, sift, and fold are all terms that the person making the cake would need to understand.
- The method consists of a sequence of steps that the user must follow in the correct order to achieve the desired result – it would be foolish to put the cake tins in the oven before making the mixture.
- Some steps require the user to make comparisons and take decisions based on what they find – in this case the user must repeat the action "beat" until the mixture is white.
- Some steps require the user to make choices – here the cake is covered in chocolate only if desired.

It would also be possible (but admittedly unlikely) for a cook to follow all the steps of this recipe without any knowledge of what the final result was going to be. That is exactly how a computer interacts with a program; it follows the instructions that it is given without any knowledge or understanding of the result that the programmer is trying to achieve.

A computer program is simply a "recipe", called an *algorithm*, presented to a computer in order to allow it to carry out some task. The recipe is written by a programmer and is expressed in a programming language. The programming language has a vocabulary from which the programmer constructs the instructions that the computer can understand and execute. These instructions form a computer program.

Programming languages

Early "computers" were designed to fulfil a single purpose. The devices generally considered to be the earliest computers, the Difference Engine and Analytical Engine, designed and partially built by Charles Babbage, were intended to carry out the calculation of mathematical tables. This repetitive task was an obvious candidate for automation, especially as the many errors in the human-generated tables could prove very costly.

Babbage never completed his machines. He was misunderstood and ridiculed during his lifetime and died unknown. One of the few to recognise the potential and importance of his work was Lady Ada Lovelace, now generally recognised as the first computer programmer. Lady Lovelace developed a mathematical notation for representing different ways to "program" Babbage's Analytical Engine. Sadly, as the machine was never built, it was never programmed. Today there is a replica of Babbage's Difference Engine in the Science Museum in London. It is the first exhibit in the gallery tracing the history of computing. Any self-respecting programmer should take a moment when in London to examine the Difference Engine.

The first electronic computers were programmed in what we would now consider very "low-level" ways. The programming involved changing physical components in the computers: levers were set, wires were moved and switches were changed. This was a slow and error-prone job. This was also the time when the first computer program "bugs" were encountered; these

were moths that crawled into the computers and promptly died, and thus stopped them working[2].

It became apparent that a different way of programming computers was needed. The existing way of working was too close to the machine's way of operating; the programmer was being forced to think at too low a level. This led to the development of the first programming languages expressed in a form that was more convenient to human programmers.

This was an improvement, but it was not ideal. Every computer had its own language, and programming any computer was still a complex and time-consuming task. A language was needed that would be easier to use and that could be used on any computer. A "high-level" language was needed.

This idea had been seen before. At the end of the Second World War, Konrad Zuse had developed a high-level language that he called Plankalkul. This was never implemented, but it remains as the first high-level programming language ever described.

The first language to be implemented and used was FORTRAN, released in 1957. FORTRAN was designed mainly for mathematical applications and was highly portable; FORTRAN programs would run unaltered on many different types of computer. Languages based closely on the original version of FORTRAN are still popular for many mathematical applications today.

Many more languages followed. Some were designed to be truly general-purpose and others were designed for particular types of application. Notable languages include:

Lisp	Lisp was designed specifically for List Processing applications. It is still used today in some areas of artificial intelligence.
Algol	Algol offered programmers the ability to express algorithms in a neat and concise way. In many ways it is the earliest ancestor of C++.
COBOL	COBOL was designed for applications in business. It offered sophisticated features for processing large files of data to produce the sorts of reports that managers needed. Decisions made in the development of COBOL programs in the early years of computing led to most of the panic surrounding the "Y2K Crisis" at the start of this century.
BASIC	As programming languages developed there was a need to train new programmers. BASIC was a language designed for beginners; it included all the features of most other languages and was a fine learning tool. Most of the early home microcomputers in the 1980s provided BASIC as their programming language.
Simula	Simula was a special-purpose language, designed to carry out simulations in applications such as queuing theory. In a way it is also an ancestor of C++, as it introduced the concept of an object.

2 The moth normally stopped working too. There is actually a school of thought that this story about the moths is an "Urban Legend". Why not spend a bit of time looking in to this on the web?

C++

We come now to C++. C++ is the latest in a proud family of languages. Its origins can be traced back to a language called BCPL. BCPL was developed at Cambridge University in the late 1960s as an alternative to FORTRAN. BCPL was developed by Martin Richards while he was visiting the Massachusetts Institute of Technology (MIT); it became widely used by those working on the new Unix operating system there.

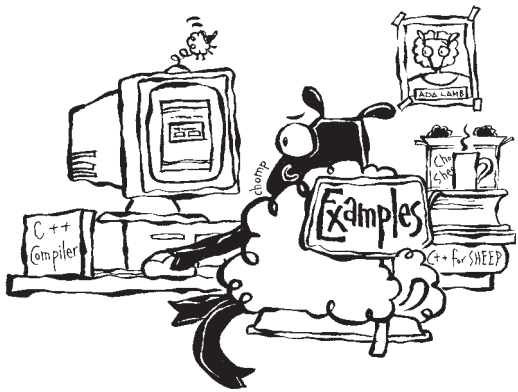
A more direct ancestor can be found in the B language, also developed at MIT in the late 1960s and early 1970s. B was designed as a programming language for developing systems on Unix, which was yet another project being developed at MIT. B was developed, extended and refined and eventually evolved into a new language, imaginatively named C. C was first fully described in *The C Programming Language* by Brian Kernighan and Dennis Ritchie in 1978. C became one of the most popular programming languages, if not the most popular, in the 1980s and 1990s.

C++ is essentially a development of C with extensions to support something called *object-oriented programming*. It was first proposed by Bjarne Stroustrup in the early 1980s. Stroustrup was influenced by Simula and wanted to extend C to include object-oriented facilities. C++ offers significant advantages over C for many applications, although even Stroustrup himself has remarked that C is to be preferred in some areas. The key is choice: C++ can be used as an object-oriented language or it can be used in a more traditional, C-like, way. This flexibility means that C++ is now one of the most popular programming languages in a whole range of applications.

C++ is the current descendant of a long line of languages. It draws from C and Simula, both of which were first developed some thirty years ago. In doing this it also draws influences from all the other languages that have gone before. Lady Ada Lovelace wouldn't recognise C++, of course, but some of the very basic ideas that she first proposed for Babbage's Analytical Engine are still in there.

C++ is not the only programming language widely used today. The closest "competitor" is Java, originally developed by Sun as a truly portable language to run on any computer. Microsoft has recently developed its own proprietary language offering many similar features, C#. In some applications, though, the old languages are still used. FORTRAN and Lisp are still used in some areas, and many COBOL systems are still around and in use. If C++ is as successful as these languages it will be around for many years to come.

The history of programming languages is quite fascinating in its own right. There are pointers to help you find out more in the Further Reading section at the back of the book.



There are no real examples in this chapter (I did say that some chapters wouldn't have any!), but here's an illustration of something that you should remember while you learn to program:

All programming languages are basically the same.

This is simple to illustrate. A first example is something called a loop, and in particular a "for" loop. Look at these for loops written in a few programming languages:

```
Pascal  for i := 1 to 10 do
BASIC   for i = 1 to 10
C       for (i = 0; i < 10; i ++)
C++     for (i = 0; i < 10; i ++)
```

Take it from me that each of these four has pretty much exactly the same effect in a program. They are examples of the same type of command in four languages designed for quite different purposes. Even if you don't yet understand what a `for` loop achieves, you should be able to see the similarities (yes, the C and the C++ are the same).

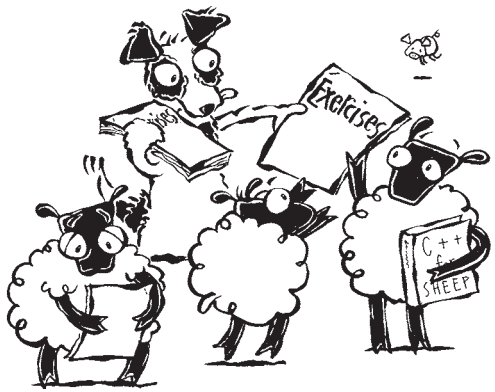
Computer programming languages *evolve*. When someone designs a new language, they include all the basic features that are found in other languages, and it makes sense to use a way of expressing them that is similar to these other languages; for one thing it makes the new language easier to learn and more likely to be adopted.

Here's another example. This is a conditional statement (a statement that lets a program make a choice), in this case testing whether some value is less than 10:

```
Pascal  if x < 10 then
Python  if x < 10:
C       if (x < 10)
C++     if (x < 10)
```

Once again the C and C++ are the same, which is to be expected as C++ is a development of C (in fact, all valid C programs are also valid C++ programs). But the examples in the other two languages look very similar, as would examples in many more languages.

Remember this. You are not just learning C++. You are learning to program and you are learning principles that you will be able to apply in many other languages.



1.1 If you have access to the World-Wide Web look up some information about some of the main figures in the development of C++. Dennis Ritchie, Brian Kernighan and Bjarne Stroustrup all have homepages that can be found quickly and easily. There's a list of web search engines at the back of the book.

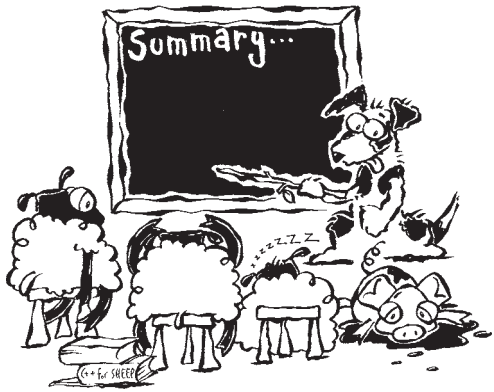
1.2 An important part of your development as a computer programmer is that you know something about other figures whose work has been fundamental to the development of programming. Use the web to find out more about Charles Babbage, Lady Ada Lovelace, Alan Turing and Grace Hopper.

1.3 In this chapter a recipe for chocolate cake was shown as an example of an everyday thing that has some of the same characteristics as a computer program. List at least two other everyday things that have the same characteristics.

1.4 Look around you now. What things around you have computer programs inside them? Who do you imagine wrote them, and how?

1.5 If you have access to a computer running Microsoft Windows, press *Control-Alt-Delete* and click on *Task Manager*. Under the "Applications" tab you'll find a list of the programs running on the computer. Try starting and stopping a few to see what happens. There are other interesting things available to look at too.

1.6 Alternatively, if you have a Unix system, type *top* at the shell prompt and look at the output. This shows you all the processes (programs) running on the computer. Experiment with starting and stopping some to see what happens.



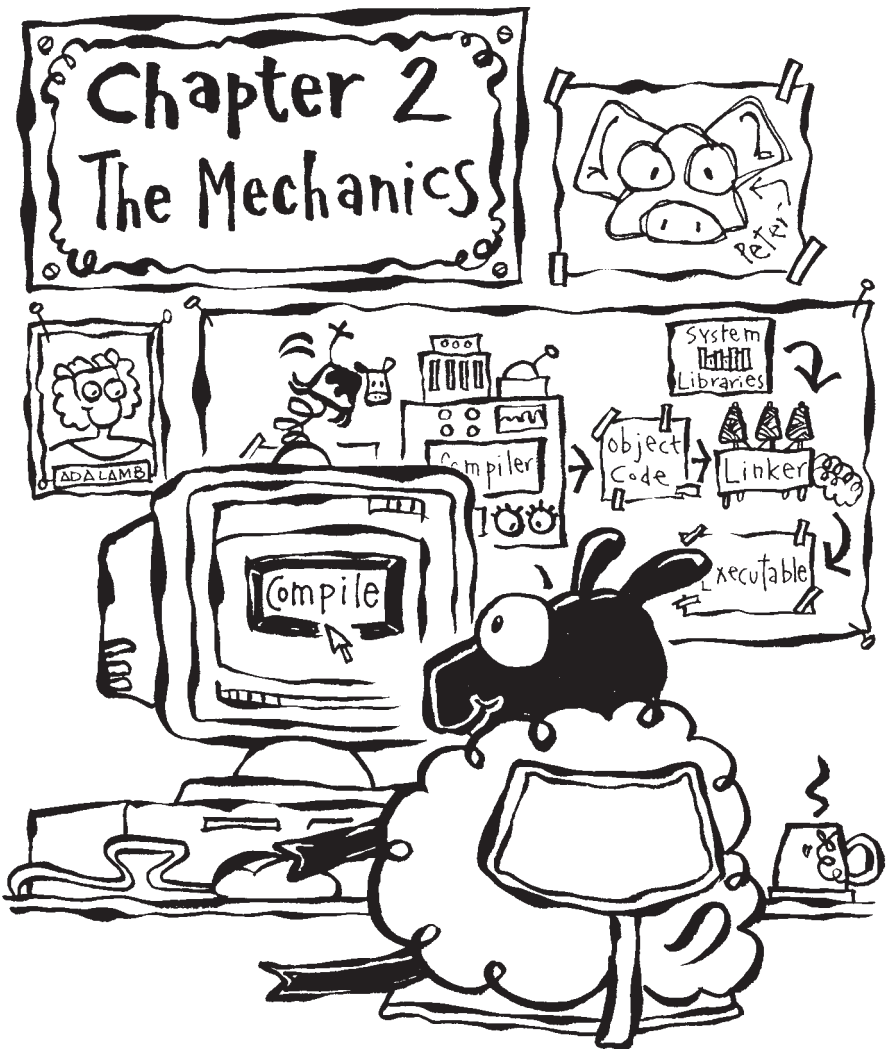
Computers are all around us and so, therefore, are computer programs. Any computer that is carrying out any useful task has somewhere inside it a program. In fact, it probably has many programs. All these programs were written by human programmers using a programming language.

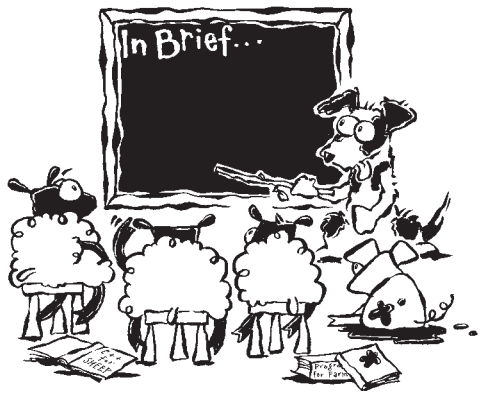
As computers have become more sophisticated, so have the languages that are used to program them. Early computers were programmed with switches and levers and then by low-level languages. This was so time-consuming and error-prone that higher-level languages were developed. Over the years some of these languages have died out, while others have been refined and developed further.

C++ is one of the latest languages in this evolution.



Chapter 2
The Mechanics

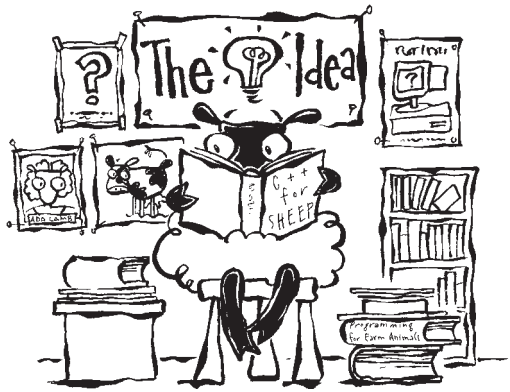




Humans and computers do not speak the same language. Humans communicate in spoken language, a very complex system of nouns, verbs, adjectives, and so on. Computers, on the other hand, are much simpler. Their "language" has only two symbols, 0 and 1. Obviously some translation must take place if humans and computers are going to be able to communicate effectively. Something will have to happen if a computer is going to understand what a programmer is asking it to do.

A computer program is a description of a way to solve a problem or carry out a task. It is written in a form that humans can use and understand. This chapter introduces you to the processes that take place when a computer program expressed in this form is translated from this representation into a form that computers can use and understand. Both these forms are actually the same program in the same way that this book translated into Japanese would still be this book; all that has changed is the way it is represented.

After reading this chapter you should understand what a *compiler* and *linker* are and what the process of *compilation* involves. You should understand how your programs will be linked with the C++ *system libraries* to produce versions that can be executed by your computer. You should also understand how to compile and run your programs in whatever programming environment you are going to use.



The previous chapter explained that a computer programming language is essentially no more than a way for a human programmer to communicate

with a computer. Humans find it very hard to express themselves using the 1 and 0 binary language of computers[1], and computers find it equally difficult to understand natural human language. A computer programming language is something of a halfway house between the two: it's something that both humans and computers can understand. In many ways the language is not quite exactly halfway; it's usually rather closer to the human's side than to the computer's. This means that some special software is needed to translate the program into a form that the computer can execute. This translation is a process called *compilation*. This is the process that is explained in this chapter. But first a word about the computer environment that you're going to use.

The "Local guide"

Computer systems are all different. This is a book about programming and C++ and I don't want all the details of many different computer systems to get in the way. I certainly don't want this book to be useful only to people using a particular type of computer system.

If you read the introduction, you'll remember that I'm going to have to assume that you have a document that I'm going to call the *Local guide*. This guide might be a manual that came with the system or it might be something specially written for you as part of a course. I'm going to assume that it will explain to you how to create and run programs on the computer system that you're using. I don't mind what that system is just as long as it's capable of running standard C++.

What I'm going to describe now is the process that happens whenever any program is created and run on any computer. This process is essentially the same on every platform.

Creating and naming a program

A program exists in a file on a computer system. Each file on the system has a unique name, consisting of a memorable name for the file and usually an extension:

```
memorable.ext
```

The *file extension* normally indicates what type the file is. You may well be familiar with `.doc` files that contain Microsoft Word documents, for example. Your system will probably require a particular extension that indicates that the content of the file is a C++ program. Your *Local Guide* will tell you what this is, but common extensions are `.cc`, `.cpp`, `.c++` and `.C`. The system I used to write the programs in this book requires that I use a `.cc` file extension for C++ programs, so that's what all the programs you'll see in this book will have. You might have to change the extension to use the programs on your system, but the programs themselves shouldn't require any changes.

These are possible names for C++ programs:

```
program.cpp
sheep.cc
flossy.c++
```

1 Although this is precisely what the users of early computers had to do. A switch that was on was a 1, and one that was off was a 0, for example.

The extension should never be left off. Doing so will probably cause errors. Ideally the name of the file should give some indication of the purpose of the program.

C++ programs usually have other files, called *header files*, associated with them. There are rules for the extensions of these files too; they are usually required to have the extension `.h`, sometimes `.h++`, or even `.hpp`. My system requires a `.h` extension and so that's what all the header files in this book will have.

These are possible names for C++ header files:

```
header.h++
useful.hpp
sheep.h
```

Any C++ program file is created with a simple *text editor*, or sometimes in a more sophisticated *interactive development environment*. Common text editors are `vi` on Unix systems and `PFE` or `Notepad` on Microsoft Windows systems. Many C++ systems on Microsoft Windows also provide an interactive development environment, normally referred to as an IDE; Microsoft's Visual C++ is an example.

The program prepared in this way, whether with a text editor or with an IDE, is called the *source code* of the program. It is just a file containing plain text; the computer has no idea at this stage of how to carry out the instructions in the program.

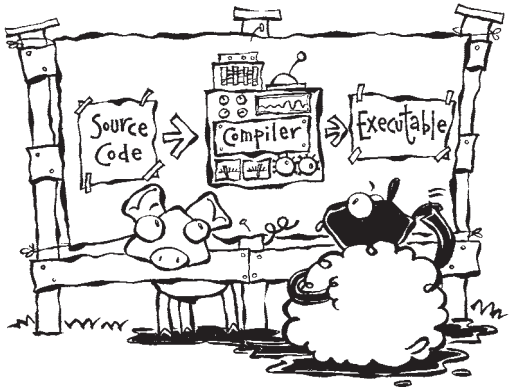
It is worth spending some time becoming reasonably proficient in using your text editor or IDE. It's normally possible to do most editing tasks with just a few commands in an editor, but many editors also have many much more powerful facilities and tools hidden away. You will spend a lot of time editing files as you work on your C++ programs, so it's more than worthwhile getting used to using the editor. Your *Local guide* should give you plenty of information.

Compilation

When the programmer has created all the source code and is happy with the program it is time for the program to be translated into a form that the computer can actually run. This is a process called *compilation*. Compilation involves the use of another program called a *compiler*[2]. The final result, a version of the program that can be run by the computer, is called an *executable* file. The executable file is written completely in a language that can only be understood by the computer; it is totally impenetrable to most humans.

This gives a process that can be represented like this (see overleaf):

2 Here's a poser for you. Compilers are computer programs. They are written in computer programming languages. So, who wrote the first compiler and what did they use? To make it worse, I'll tell you that most C compilers are written in C.



The actual process that a programmer goes through to compile a program is different on different computer systems. Popular compilers include Microsoft's Visual C++ and the Free Software Foundation's *g++*, and every one is different. Sometimes compilation involves typing a command at a prompt, and other times it is simply a case of clicking a button.

On the Unix system I used to write the programs in this book I used the *g++* compiler so to compile a program stored in the file *myProg.cc* to produce an executable file called just *myProg* I would type[3]:

```
tetley% g++ myProg.cc -o myProg
```

Once again you'll need to consult your *Local guide* to find out how to compile programs on the system you'll be using.

Compilation errors

Sometimes a program fails to compile because of some error that the programmer has made. The error means that the compiler cannot process the source code to produce an executable file. Faced with this situation the compiler will do its best to tell the programmer what the error probably is (or at least where in the program the error seems to be) and the programmer must then find and correct the mistake.

The way in which you will find and correct errors in your programs depends again on the system you are using. My Unix system simply tells me what the error is and in which line in my program the error appears, and leaves me to use the text editor to correct the error. More sophisticated systems will present the programmer with the part of the source code producing the error, with the offending section highlighted in some way. Again you should check your *Local guide*.

New programmers often have terrible problems finding errors in programs. The secret in finding them is to remember two things. First you should always be systematic; correct one error and then try to compile the program again. A small mistake in the program, even one single character in the wrong place, can cause a whole host of different, seemingly unconnected, errors. It is always a good plan to make the error that you correct first

3 The *tetley%* in this is the prompt on my computer, and isn't part of what I type to compile a program. My computer is named after the famous Joshua.

the one that the compiler reported first; a small error at the top of a program can have many knock-on effects later on. Second, remember that compilers are not especially clever and get confused easily; what they are reporting is often the first part of the program that they have failed to understand. If the line where the error is reported looks fine, always look at the lines immediately *above* the point in the program where the compiler says the error is.

Running the program

After the program has been compiled into an executable form it is ready to be run. As usual, the way in which programs are run is different on different computer systems. On a Microsoft Windows system it is normally enough to double-click the mouse on the executable's icon, or the development environment may run the program automatically. When I run a program on a Unix system I just type its name preceded with `./`:

```
tetley% ./myProg
```

It's time to consult the *Local guide* to check how to do it on your system.

Other useful things

When working on a program it is often very useful to be able to print your source code out. This gives you something that you can scribble all over as you try to work out why your program isn't working as you hope and expect[4]. You'll have to consult the *Local guide* to find out how to print your programs and to find out where the printers are.

There may be other tools available to you to help you as you learn to program. There may be a special program called a *debugger*; this is a tool that lets you step through a program a line at a time to help you find and correct problems. There may be a program that helps you format your source code neatly. Find out what's available and make sure you can use it.

A closer look

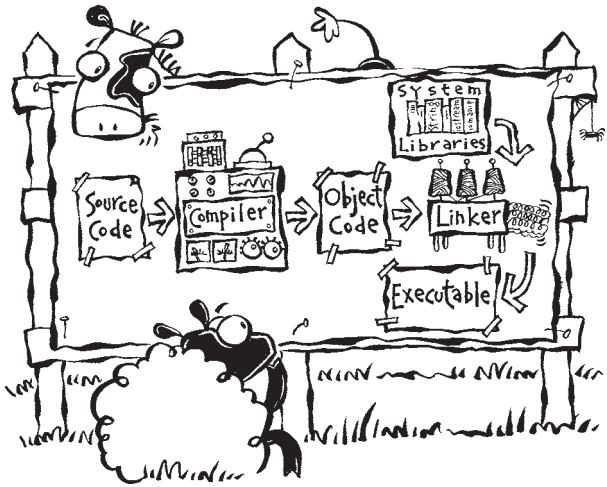
The description of compilation in this chapter is a little simplistic. It will help if you have a slightly better knowledge of what's actually going on when you compile a program. A full discussion of how compilers work is way beyond the scope of this book and is actually way more than most programmers need to know, but a little more detail will help you when you start looking for errors in your programs.

First you need to know that there is an intermediate stage between the source code and the executable code. This stage is called *object code*; this is the first thing created by the compiler from your source code. You may even sometimes see the files produced by this first stage of the process; the extension they have depends as usual on the computer system and compiler (`.o` and `.obj` extensions are common for files containing object code). The object code is then *linked* (by another program called a *linker*) with other object code from various system libraries. These libraries contain the object code for the programs that are used for the basic C++ functions (so that the program knows how to display characters on the screen, for example). All the different object code is linked together to form the final executable.

---

4 Or indeed something that you can show to people to impress them.

So the overall process looks like this:



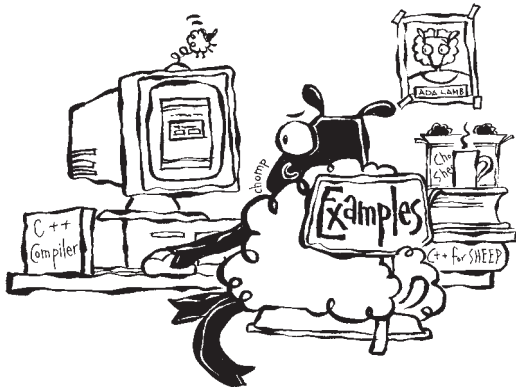
Sometimes the whole compilation process will fail at this linking stage. This might be because the wrong libraries are being linked or because some essential library has been left out. What has gone wrong should be obvious from the error messages generated.

From source code to executable

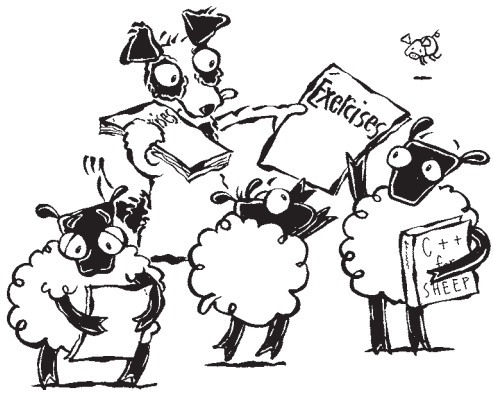
As you write and test your programs it will help if you remember to think about what it is that you are creating. You are creating a "recipe" for solving a problem and you are using a language that is meaningful to humans[5]. Before this can be run on a computer it must be translated into a form that the computer can execute. The *source code* of the program must be translated into an *executable* version of the program.

This translation is a process called *compilation*. Compilation is actually a two-stage process. First the source code is translated into *object code* and then this object code is *linked* with various system libraries. The result of this linking is the final executable – a version of the program that is very different in the way it looks to the source code that you have written!

5 Well, to some humans.



There are no real examples to work through this time. You need to check your *Local guide* and become familiar with the systems that you'll be using.



There are no exercises this time either, but there are some things I need you to do. Get hold of your *Local guide* and make sure you find out answers to these questions.

2.1 What is the required file extension for C++ programs on your system? Does it matter if it's in upper or lower case?

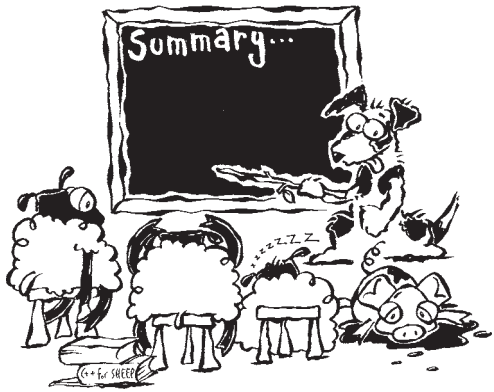
2.2 What is the text editor that you will use? Or will you be using an IDE? How do you start it? What are the basic commands you need to edit a file? Can you get a copy to install on your own computer at home?

2.3 How do you compile a program? What is the command? Where does the executable version of the file get placed?

2.4 How does your compiler present you with any compilation error messages?

2.5 How do you run an executable file? Where does any output go?

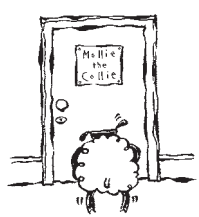
2.6 How do you print a hard copy of your programs on your system? Where are the most convenient printers?



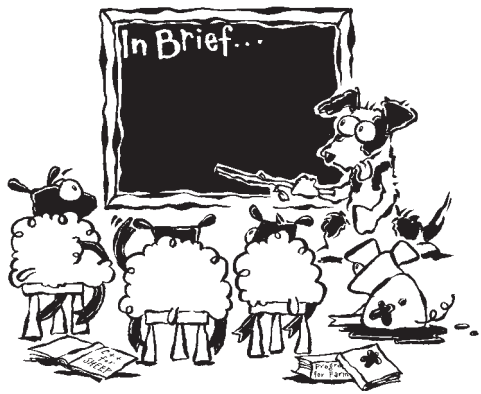
There are two steps in creating a program that can be run on a computer. First the source code must be created in a file; the name of this file will be required to have an extension that indicates that it contains a C++ program. When the program has been created it is compiled to produce an executable version; it is this version that can be run on the computer. This second process involves the use of two other pieces of software – a compiler and a linker. The compiler produces object code and the linker links this with the various C++ system libraries to produce a complete working executable version of the program.

There are several skills that you need to master in order to create and run your programs. You need to know how to create files and you need to know how to edit them. You must be able to compile your programs and correct the errors that the compiler finds. Finally, you must know how to run the executable and how to find the output. Unfortunately these skills differ depending on different computer systems; your *Local guide* will give you the details for your system.

To do any of these things you need to know how to write a simple C++ program. So it's nearly time to get on with it. But first there are some things you need to know about learning to program....




Chapter 3
Learning to Program
COW
Lesson One
History for sheep
GEOG FOR PIGS
Chair Design a

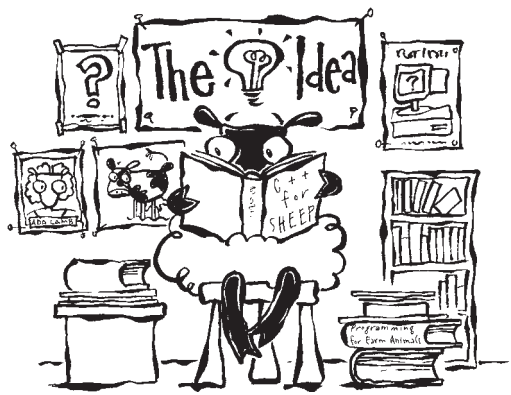


Learning to program is difficult.

Before you start reading the chapters in this book that describe the features of C++, and certainly before you start to do any actual programming, it's important that you realise that learning to program is difficult. It's also important that you think about the ways in which you've learned things in the past, and that you understand how and why you will have to adapt the ways you learn if you are going to learn to program successfully.

After reading this chapter, you should understand what you've let yourself in for! You should know why learning to program is sometimes found difficult by many people, and you should have some sort of idea about how you are going to set about learning yourself.

Don't be put off though. Just because something is difficult doesn't mean that it's impossible!



Before you start any C++, I want to explain to you what learning to program is all about. You need to know what it is that you've taken on and you need to understand something about the skills that you're going to have to call on as you learn to program. You're also going to have to think about and understand something about how you learn. Learning is a very personal thing and you are going to have to think about the best way for you to learn to program.

I'm going to start with a quote that could well sum up what you've taken on.

You have chosen a dangerous and challenging path, Adventurer.

A long time ago I used to play a computer game. It was an adventure, and it just used plain text. These were the days when computer games involved nothing more sophisticated than typing instructions to the character you were supposed to be playing. There were no graphics in those days, and certainly no Lara Croft. The computer didn't even have a mouse, and we got very excited if a program used some sort of colour beyond different shades of grey. This game involved all the usual things that you'd find in games like this. You'd move around mazes of corridors, battle hideous demons, give small pots of gold to passing dwarves, and fall down the occasional bottomless pit – you get the idea. Much the same happens in many games today, only with better graphics.

I still remember how this game started. After it had loaded (something that in those days took the better part of half an hour) the screen cleared and this message appeared:

You have chosen a dangerous and challenging path, Adventurer.
You are at a crossroads. Paths lead North, South, East, and West. There is a sword and a scroll here.
What next?

You can guess what happened next. I would dutifully pick up the sword and scroll. I would read the scroll and ponder the pretty useless clues that were on it. Then I quickly rushed off in a random direction. I very soon met the fire-breathing dragon that didn't seem to like me very much, said the wrong thing to it (hitting it with the sword never seemed to help), and was promptly eaten alive or burned to a crisp or both. Then it was back to the start and I started the whole process again.

I never finished this game. I never did work out which dwarf desperately wanted the bucket with the hole in that I found behind the old oak tree. The stone tablet bearing mysterious runes remained a totally useless rock with some odd scribbles on it as far as I was concerned. My friend finished it though. He spent hours on the computer going east, west, down tunnels, bargaining with trees that talked, and many equally strange things. Eventually he found whatever it was he was supposed to be looking for and won the game. He succeeded and I did not.

You are reading a book about learning to program. This shows that you too have chosen a challenging path. I have to admit that it's not very likely that you'll meet a fire-breathing dragon, a talkative tree or a dwarf with a bucket fixation (all good metaphors break down eventually), but you're certainly in for an interesting and challenging time.

Let no one convince you that this will be easy. Learning to program isn't easy, and anyone who tells you that it is easy is just plain wrong. At the same time, learning to program is not impossible. It can't be, or there wouldn't be any programmers! There's nothing special about people who can program. As you read the rest of this book, just remember this. I can program reasonably well and I'm not especially clever. If I can do it, so can you!

Think about how you learn

One of the reasons why learning to program is so difficult is that it's such an unusual subject. It's unusual because it's unlike most, if not all, of the subjects that you've met before. You've probably learnt many things from your

teachers in many different subjects before now. This means that you have developed a set of study skills that you use; you have a set of ways in which you like to learn and in which you know you learn well. As you approach this new challenge, you'll have to develop a whole new set of learning skills. I'm afraid the old ones just won't do any more.

Read this paragraph and then stop reading for a minute. Take a break and think about how you learn best. Do you learn best alone or with others? Do you learn best in the morning or in the evening? Or do you learn best when you're in a particular place? Do you learn best in the quiet of the library or in your own room with your own music playing? Do you like to experiment with something new until you understand it or do you need all the facts in front of you before you start? Think about this now.

Welcome back. After thinking about the way in which you like to learn, you have to realise that some of this just isn't going to work now that you're learning to program. Some of it will still work, but a lot is going to need to change. If you like to learn with others you'll find that this is a fine way to learn to program. Programming is usually quite a social occupation and is often done in teams. If you like learning by experimenting you're also off to a good start. Programming is a practical subject, and you'll certainly need to spend a lot of time "doing" it. On the other hand, if you prefer reading books you're going to have to change a bit. Learning to program is not like, say, learning a set of dates in history or the vocabulary of foreign language.

Let's think about why this is.

Programming is like driving

Imagine this. Suppose this was a book about learning to drive a car, and that you'd never driven a car before. I would write chapters about using the brakes, steering, changing gear and so on. Imagine that you'd read the book from cover to cover. You might even have memorised parts of it. You might even have passed a test on the contents. Now imagine that I put you inside a car in London and told you to drive to Inverness. I (and you!) would be very surprised if you or the car arrived in one piece. I hope we agree that no one would seriously consider learning to drive a car like this.

People learn to drive a car by practising. They spend a lot of time sitting next to an experienced driver who tells them what to do and helps them out occasionally when things seem to be going wrong. They drive around and practise the skills they need to develop and perfect. When they've developed these skills sufficiently they take their driving test and (hopefully) they pass. Sometimes they don't pass and they need to practise some more. Some people find learning to drive harder than others.

Learning to program is very like learning to drive a car. You are learning a *skill*. You will need help from people who already have the skill; hopefully, one of those people will be me! Above all you will need to practise. There's no way I can teach you to program if you just read this book, but I can teach you if you read the book, then try out the exercises, and then go and practise some more. With any luck you also have some more people who are going to help you. Whatever you do, don't be afraid to use them!

But there's more. Some people pass their driving test the first time. Others have to take it many times, but most people pass eventually. There are very few people indeed who could never pass a driving test if they really put their minds to it and practised. Programming is just like that. Some people take to

it straight away, and they're the lucky ones. Most people take a bit longer, but anyone can get there in the end. I've never met anyone that couldn't learn to program. Not if they tried.

If I had taken the same attitude to learning to drive as I took to my adventure game all those years ago I would never have passed the driving test. The game was difficult so I lost interest and gave up. Then again, looking back, I probably believe that I could have completed it if I'd really wanted to and if I'd carried on. Learning to program is like that too; if you give up when you find something difficult you'll never succeed and learn. But if you persevere and if you're determined to succeed you will do. I've seen many people struggling as they learned to program, but the only people I've seen fail are those that gave up. Don't be like them!

What you need

Before you start looking at some C++ we need to check that you're prepared. There are some things you're going to need, so let's quickly run through them.

You will need a book or two. You have one in your hand now, but you also need something that describes how your local computer system works. I've called this your *Local guide*. If you're learning to program as part of a course you'll probably also have something else that contains the exercises you're going to complete as part of the course. You'll also need a reference to remind you of the syntax of C++ as you program; you can find just such a handy reference in the back of this book.

You obviously also need a computer. You need something that will run your programs and will show you the output that they produce. For writing the C++ that you will meet in this book any sort of computer will do; you don't need anything especially powerful. A printer so that you can print out your programs and draw all over them would help, but it's not vital.

Your computer needs to have a C++ compiler; you learned what one of these was in the previous chapter. You need to know how to get the compiler to compile your program (most compilers will take care of the linking phase too) and you need to know how to execute the compiled version. There are a great many compilers available all of which are very different to use. As I said in the last chapter, I'm going to have to assume that you have a *Local guide* that explains how to use your compiler and the rest of your system.

You will also need a text editor to create the files that will contain your programs. This might be part of the system that includes your compiler, or it might be a separate program. You will need to know how to use your editor to create your programs and how to get them from the editor to the compiler. Once more, there are lots of editors that you might have available, so I'll have to assume that your *Local guide* explains all the details of that.

If you are missing a compiler or an editor there are plenty of places where you can get them without spending any money. See the Further Reading section at the back of this book for a list of web sites where you can download some pretty good free compilers and editors, and for a way to get hold of them if you don't have access to the Internet.

You will also need access to an expert or ideally to more than one! This is someone who can already program and who is ready and willing to help you. There will be times when you just can't work out why your program is producing the wrong answers and you need someone who can help you work

out what's wrong. You might not need help very often, but most new programmers will need some help and advice at some point while they learn.

Above all you need a willingness to learn and you need determination. You need to be sure that you want to do this, and you need to be determined to see it through.

If you've got all these things then read on!

Getting help

If learning to program is difficult then learning to learn to program is equally tricky! The most important thing is that you need to know when you should get help from one of your experts; get help too soon and you'll end up not learning anything, but leave it too late and you run the risk of not finishing your program in time. Knowing when you should get the help you need is a tricky business.

At some point in the not too distant future you'll be sitting in front of a computer trying to write a program and you'll find that you're "stuck". You might have some sort of bizarre error message that you just can't work out, your program might be insisting on producing the wrong results, or you might not have the vaguest idea how to complete the next part of the task you've been set. Whatever the problem is, you need to do something about it.

Staying in front of the computer typing random symbols or copying random sections of programs from your books for hours on end is unlikely to work. It just might (after all, an infinite number of monkeys typing on an infinite number of typewriters would eventually produce this book!), but it's not very likely. You need to do something more positive. I often find that the best plan is to walk away from the computer and do something totally different for a while. You would be surprised how often the solution can suddenly pop into your head when you least expect it as you stand in the queue for a coffee! But if it doesn't pop you need to do something else.

There are many things you could try. You could try re-reading the relevant chapters of this book. If you've already done that, or if a re-reading doesn't seem to help, the collection of examples at the end of the chapters might help. You could look at some programs that do something similar to what you want to achieve and try to adapt them.

If you don't get anywhere after this, go and talk to one of your experts. They should be able to tell you what's wrong and they'll probably be more than happy to do so. At the same time you should resist the temptation to let your expert tell you the answer or write your program for you. If you let them do that you'll learn nothing.

The golden rule for getting help when you're learning to program is:

Try to work it out yourself first, but don't be shy about going to an expert.

Always remember that the experts all learned to program for the first time once. The advantage they have over you is just that they learned a while ago and that they've had a lot of practice since then. They've seen all the errors before, and they can remember what it's like for you. Never ever believe that they're more intelligent than you; they've just got a skill that you haven't (yet!).

Listen!

I spend a lot of time talking to many students as they learn to program. Sometimes I ask them what advice they would give the students that follow them in

this adventure. What do they wish someone had told them before they started their course? These are some of the things they say again and again. Listen to them.

"Don't leave it to the last minute!"
In some of your subjects you might be able to get away with cramming just before the exam. This approach might even have worked for you in some subjects before. This is an absolutely disastrous strategy to use now that you're learning to program. Work steadily, work through the examples you're given, and keep up. The sooner you start your programming exercises the sooner you'll get stuck and the sooner you'll be able to get help from your experts.

"Go to the lectures!"
This might seem like an obvious point but it *has* been suggested that sometimes some students don't go to all of their lectures. Once more, you might sometimes be able to get away with skipping a few lectures in some subjects but you're not going to be able to skip classes in programming. The problem is that programming will be presented to you as a series of concepts, and each one builds on the ones that came before. So if you miss a lecture or a class you'll find that you don't understand the next one, and so it goes on. You might even find that you can never catch up properly.

Of course, you learn to program by writing programs and by practising, but the lectures in your course still have an important part to play. They explain to you the new concepts and ideas that you'll come to use and will provide you with useful examples. You build on what you see in the lectures to develop your own understanding and programming skills. The lectures should also be a valuable opportunity to ask questions of one of your experts or to listen to questions from other students.

Missing a crucial concept as you learn to program can be educationally fatal. Educationalists have a name for the feeling that this can cause in people; they call it "learned helplessness". Basically this means that someone decides that they "just can't learn to program" or "just can't do maths" as an explanation for their failure to do just that. Even though they decide and believe this, it isn't really the problem. The problem is simply that they are missing some basic piece of knowledge. More often that not, this is simply because they skipped the class in which it was first explained. If you ever feel like this you should seek help from one of your experts as soon as possible. Obviously, though, the best plan is never to get to this state.

"Practise!"
While lectures are important, you can't learn to program just by attending them. Nor can you learn to program just by reading a book. Programming is a practical hands-on skill that you learn by doing. Remember that just because you can understand a lecture and the corresponding chapter in this book you can't necessarily apply what you have seen to a problem that you haven't met before. You learn to use and understand it when you get down to some serious practice.

"Get help!"
As I've said, knowing when to get help is difficult. Some of my students tell me that they later realise that they spent far too long trying to work things out themselves before getting help. Others sometimes confess that they

asked too soon. But they all agree that getting help when you need it is essential. Don't ignore the problem and don't believe it will go away!

"Buy a book!"
This might seem an odd thing to say, especially as you've (hopefully!) already bought this book. But you *will* need a book, even if you only use it as little more than an occasional reminder or reference. Even experienced programmers write programs with a reference book by their side.

"Stick at it! It's not impossible."
This last one almost speaks for itself. If (or when) you find that learning to program is difficult, remember that most people get there in the end. The people that don't are the people who give up. Don't give up.

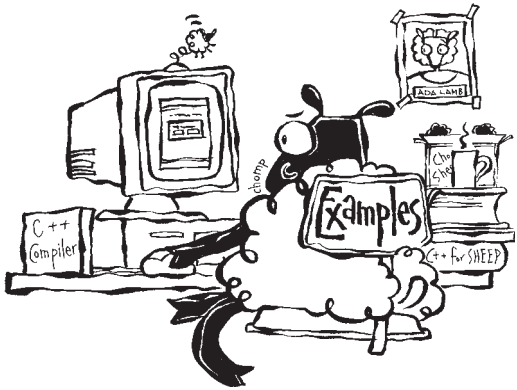
Don't panic!

I'm sorry if all this sounds very negative and off-putting. Please don't take it the wrong way.

I want you to understand that learning to program is difficult and that everyone who does it gets stuck at some point. What you have to do if you're going to succeed is change the way you learn a little. You're going to have to do a lot of work as you practise on your own. In short, you're going to have to take responsibility for your learning.

Of course, you might be one of those people who finds programming very easy and wonders what all the fuss is about. You might be a "natural" programmer. If you are then I'm sorry for having worried you unnecessarily in this chapter. But I hope you'll do me a favour. Go and find someone who's finding this really difficult and give them a helping hand. Don't finish their program for them and don't tell them the answer; if you did that they wouldn't learn anything. Point them in the right direction, explain an error message to them, or give them a clue how to finish their program. Remember that programming is a social thing. Programmers work as a team and help each other out. You'll probably enjoy lending a hand and the person you help will be very grateful.

Now it's time to start on some programming. You'll get the chance to find out whether or not you're a natural programmer!



Since you haven't done any programming yet there still aren't any examples as such in this chapter. While we're here though, let me just tell you how I learned to program.

A long time ago, when home computers were a very new thing, you could buy magazines that contained nothing more than listings of programs. You could type these into your computer (they were almost always games and were usually written in a dialect of a language called BASIC) and sometimes they even worked. I learned to program by typing these programs into my[1] computer. Of course, at first I understood practically nothing of what I was typing. For example, I remember looking very hard at this line:

```
10 LET X = X + 1
```

and wondering how this could possibly be right. I mean, *X* can't possibly equal *X + 1*, can it? In time I came to understand parts of the programs I was typing and even that one line became less of a mystery. If you can already see what it does you're doing better than I did! Eventually I could write my own programs, at first mostly by adapting those I had seen before.

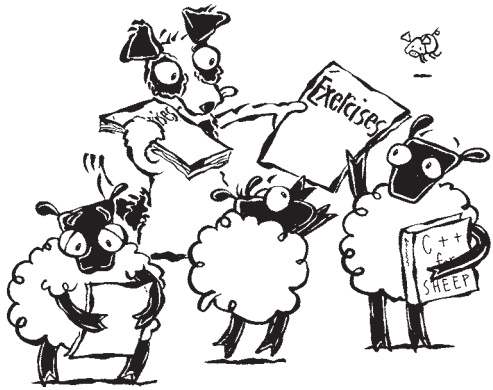
Later I took a more formal programming course. There were some things in it that I found difficult and some others that came much more easily. I got by well enough, by going to the lectures and practising in the labs. There were still some things that remained pretty mysterious and that I never properly understood, though.

I *really* learned to program when I finally got a job as a programmer. I worked eight or more hours a day doing nothing but programming. I worked with programs that controlled massive test machines in a truck engine factory; if I made a mistake the machines could have exploded and done someone a nasty injury. That was when I finally became a good programmer.

I'm still learning now. I learn new ways of doing things or new features of languages. No programmer ever stops learning.

---

1 Actually, I never had a computer of my own. It was my friend Mark's computer and we learned together. Learning to program can be a very social activity.



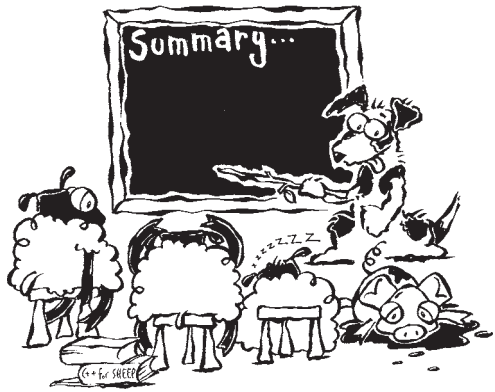
3.1 You are about to start to learn to program. Have a quick look through the rest of this book and through your *Local guide*. Plan how you are going to spend your time, and write down your plan. Are you going to read a chapter a week or one a day? How much time can you spend on practice? If you're following a lecture course, how are the lectures structured? What topics will you be covering, and when?

3.2 If you have the chance, talk to someone who can program. Find out how they learned and ask them what they remember as being especially difficult or easy. Why not ask your lecturer?

3.3 Write down some things that characterise the way in which you think you learn best. Think about things like time of day, day of the week, place, company and environment. Think about how you will make sure that you have the best possible environment in which to learn to program.

3.4 Look again at your ideas about how you learn. What are you going to have to change now that you are learning to program? For example what time of the day or week will be best for you to practise? Where will you practise?

3.5 Finally, take a look at the list of the things you're going to need. Make sure that you've got them all and that you know how to use your editor and compiler. I can't help you with this, but your *Local guide* should be able to. I'm going to have to assume that you've done it!



Learning to program is difficult but it is not impossible. It's going to require an approach to learning that will be different from the one that you're used to. You're going to have to change the way you learn; in particular, you're going to have to do a lot of practice.

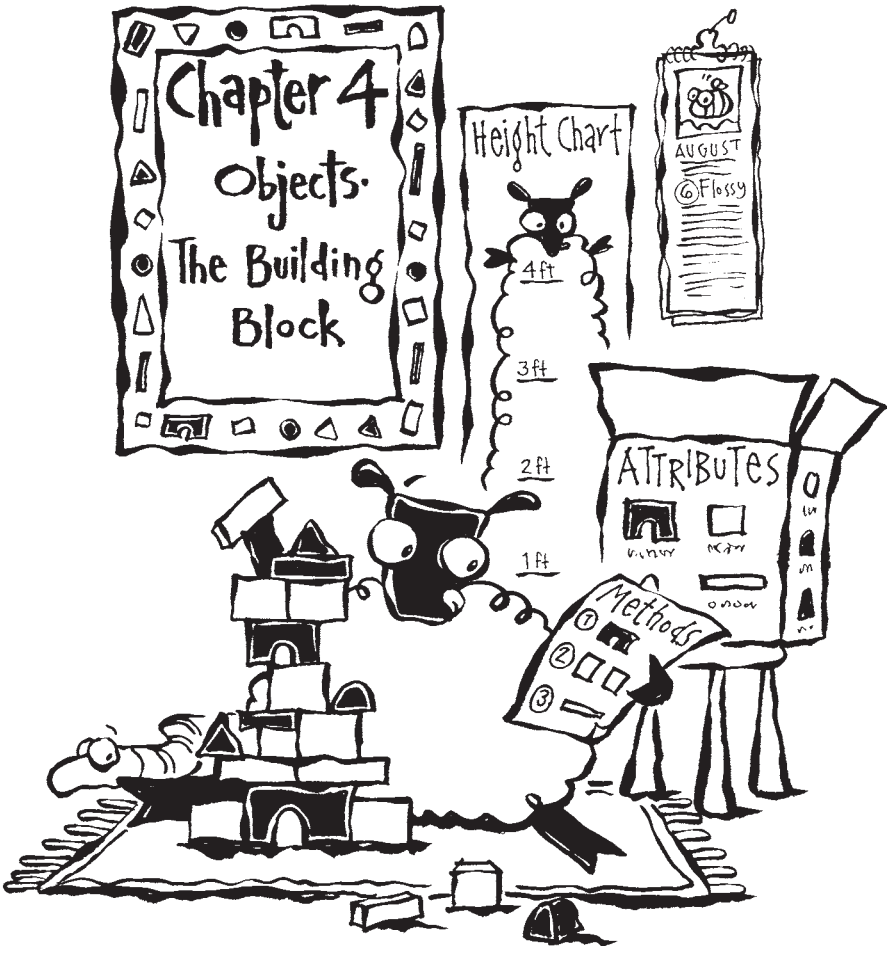
A long time ago, when I was learning to program at university, the author of the book we used[2] (a wise man called Doug Cooper) wrote in the preface that:

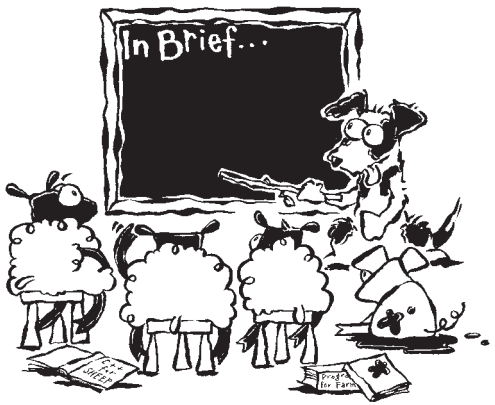
> *Now, when I lecture I encourage any student who isn't so confident to make a smart friend, and to stick by her side for the term. After all, that's how I survived my own first programming class. When I write, I try to be that friend...*

I can't think of any better final advice to give you now than that. That book got me through my first programming course too, and I still use it now, even if it is a bit battered. Let's hope this one does the same for you. If you can make a smart friend that's even better!

Now you understand what learning to program is all about let's go and learn to program!

2 That book was called *Oh! Pascal!* and was, as you might guess, about the Pascal programming language. Pascal isn't as popular as it used to be, but if you ever need to learn it, this is the book you need.


Chapter 4
Objects:
The Building
Block
Height Chart
4ft
3ft
2ft
1ft
AUGUST
@Flossy
ATTRIBUTES
Methods
1
2
3



You already know that C++ is an *object-oriented* programming language. Many other popular programming languages, including Java, Python and C#, also use this object-oriented approach. This means, not surprisingly, that programming in C++ is based around the use of *objects*; objects are the basic building blocks of any C++ program. One of the key advantages claimed for object-oriented programming is that the programs developed to implement an object required for use in one problem can be used again in another problem; the programs to implement the object are *portable*. This potential for code reuse is one of the reasons why object-oriented programming is currently so popular; code reuse has the potential to save a lot of time and therefore money.

This chapter introduces you to the idea of an object. After reading this chapter you should understand that objects are the basic components of any object-oriented computer program and you should also understand what exactly object-oriented means. You should understand how an object is characterised by its *attributes* and its *methods*, and you should be able to suggest likely attributes and methods for objects that might be found in some everyday applications.



Before starting to write object-oriented computer programs you obviously need to know what exactly an *object* is in this context. Informally an object is

something of interest in a real-world problem. When a computer program is written to solve this problem an object becomes something that is processed or manipulated by the program. The object has values of interest that are processed to produce the required results. Objects are sometimes called "entities", especially in the area of database design[1].

Objects have values that describe them, called *attributes*. An attribute is simply some property that characterises an object; people have attributes, such as name, height, shoe size and so on. In object-oriented programming some attributes can be procedures that use the values associated with the object to calculate some other value or to achieve some effect; these are called *methods*. An object is defined by the values of its attributes and the behaviour of its methods.

More formally an *object type* corresponds to a group of objects in the real world. Examples might be sheep, pop singers or people called Tony. Each object type has a particular set of *object instances*, for example Flossy, Britney Spears and Tony Blair. An object type has attributes and methods and each instance of the object type has particular values for each of these. Normally, when taken together, the values for an object instance uniquely describe that instance.

Let's look at these ideas in more detail.

Attributes

Objects are all around us. You're holding one in your hand now[2]. The object you are holding is a book. Imagine that you were trying to describe this book to someone else so that they could find it in a library full of similar books. You'd probably tell them the title, maybe the name of the author, and perhaps you'd also describe the cover. You would tell them some things that were special about this book; ideally you would tell them the things that uniquely identify it among all other books. You would not tell them that the book is printed on paper because all books are like that; you would concentrate on the things that make this book easily identifiable among all other books. In object-oriented terms you would be telling them about the values of this book's *attributes*.

An object's attributes are the features that identify or characterise it. A book has a title, an author, an ISBN and many more attributes (look inside the front cover). If you give someone the values for these attributes for a particular book they can always find the precise book you mean; the attributes uniquely determine and describe the object.

You are an object too. You have attributes that describe yourself; you have a name, an address, a birthday and many more. You are an instance of the object type "Person". So are all other people, but they have different values for their attributes. It is highly unlikely that there is another person who has the same set of values for all your attributes, or at least that we couldn't think of a set of attributes that would distinguish you from all other people.

To summarise:

---

1 You may have done some entity-relationship modelling. Objects and entities as used in this sense are much the same thing.

2 Or else you're reading this in a very strange way.

Attributes of a book	Attributes of a person
Title	Name
Author	Date of birth
ISBN	Address

You can probably think of several more possible attributes to add to each list. It is important that each attribute has one and only one value for each object instance. People do not have two names[3] and books don't have two titles. No attribute of an object instance should be allowed to have more than one value.

As well as a name, each of these attributes has a *type*. There is a particular type of value that each attribute can take; a book's title is a string of characters, a date of birth is a date, and so on. Each attribute has exactly one value and, in most programming languages, that value always has the same type.

Programming languages support a range of possible types of this sort; the names used are slightly different in different languages, but the basic types are the same. Common ones are:

Integer	Whole numbers, positive or negative – in C++ `int`
Floating-point numbers	Numbers with decimal parts, again positive or negative – in C++ `float`
Character	Single characters, generally anything found on a standard keyboard – in C++ `char`
String	Sequences of characters, normally spelling words or similarly meaningful sequences – in C++ `string`
Boolean	True or false values, named after their use in Boolean logic – in C++ `bool`

Whenever an attribute is identified it should also be possible to determine the attribute's type.

Let's look again at the attributes of a person and add in the type of value for each. The precise list of attributes for a *`Person`* object type would probably depend on the purpose of the program that was going to use it, but some common ones would be:

Person	
Attribute	**Type**
Name	String (`string`)
Gender	Character (either 'M' or 'F') (`char`)
Age	Integer (`int`)
Date of birth	String (`string`)
Height	Floating-point (in metres) (`float`)
Weight	Floating-point (in kilograms) (`float`)
Address	String (`string`)

3 *Name* here means, of course, forenames and surname (or family name) together.

This is far from a complete list; you can probably think of some more attributes to add. Whatever attributes are added each can have only one value of one fixed type.

Methods

Look again at the type of the "Date of birth" attribute in the *Person* object. This proposes that the date is stored as a string of characters. It is indeed possible to use a string to store a date, and there are many possible formats that might be used. The format that would be preferred would depend on factors such as how and where the date was to be displayed or the conventions of the country where the program would be used. Examples might be:

```
2nd January 2001
2-1-01
2/1/2001
1/2/2001
```

which could all correspond to the same actual date.

The scheme of storing a date as a string would work if the only operation ever required on a date were to print it out. Complications would quickly arise if the date were to be processed in some way or printed out in a different format. Even simple tasks such as finding the next date or a person's 21st birthday would be extremely difficult if the date were available only as a string. For anything but the simplest problems the string representation for dates would be far too inflexible. Something much more powerful is needed. The solution is quite simple; the date attribute of *Person* is in fact another object, one that can store dates. An object's attributes may themselves be objects.

Let's consider what the attributes of a *Date* object type should be. The simplest scheme is to have three integers as attributes; one each for day, month and year:

Date	
Attribute	**Type**
Day	Integer
Month	Integer
Year	Integer

There are other possibilities. Many computer systems handle dates by storing the number of seconds since some known fixed point in time; this is a simple scheme and is cheap for a computer to store. This internal representation is converted to a more human-friendly format whenever it is displayed. The advantage of either of these schemes is that the date is stored in a simple neutral form and can then be displayed in whatever format the user chooses. The simple storage makes it easy to develop programs that manipulate dates.

Displaying the date is an operation that uses an object. This operation is obviously closely linked to the object and in a way it is also an attribute of the object. It is, however, obviously a special kind of attribute. Such attributes are called *methods*. A method can be thought of as a procedure that uses the attributes of an object to produce some useful result or effect. Possible methods for a *Date* object type might be:

- Find the next date
- Find the next date and change the values of the object's attributes to the values of the next date
- Find the day of the week of the date
- Determine whether the date is valid
- Display the date in the format DD/MM/YYYY
- Display the date in a format similar to "2nd January 2001"

Like attributes, methods always have exactly one value; this is the value that the execution of the method produces. It is called the *return value*. This return value also has a type; this is usually called the *return type* of the method. Methods always return a value even if the value is "nothing at all". The idea of returning a value of "nothing at all" might seem odd, but it does make a lot of sense. The fact that a method always returns exactly one value is very consistent and easy to deal with. It is much less complicated than dealing with the possibility that a method might or might not return a value. The value "nothing at all" is represented in C++ as `void`.

A more complete specification for the methods of a *`Date`* object could be:

Date methods

Method	Return value	Return type
Find the next date	The next date	Date
Find the day of the week of the date	The day of the week	String (or perhaps an integer)
Find the next date and change the values of the object's attributes to the values of the next date	Nothing (the values of the attributes are changed, but nothing is returned)	Null ("nothing at all")
Determine whether the date is valid	True if the date is valid, false otherwise	Boolean
Display the date in the format DD/MM/YYYY	Nothing (the date is displayed, but nothing is returned)	Null
Display the date in a format similar to "2nd January 2001"	Nothing	Null

This is far from a complete list, and once again you can probably think of several more possibilities. The precise list of methods for a *`Date`* object type to be used in a program would depend entirely on the functions required from the program.

Describing an object

Attributes and methods together define an object type. For example, a full definition of our *`Date`* object type might be:

Date		
Attribute	**Type**	
Day	Integer	
Month	Integer	
Year	Integer	
Method	**Return value**	**Return type**
Find the next date	The next date	Date
Find the day of the week of the date	The day of the week	String
Find the next date and change the values of the object's attributes to the values of the next date	Nothing	Null
Determine whether the date is valid	True if the date is valid, false otherwise	Boolean
Display the date in the format DD/MM/YYYY	Nothing	Null
Display the date in a format similar to "2nd January 2001"	Nothing	Null

This definition describes all the attributes and lists all the methods available.

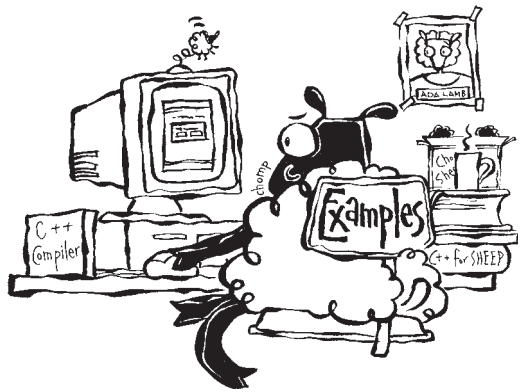
From the point of view of a programmer writing a program that would use this object type, this definition specifies the names and types of the values stored in any instance of the object type, and describes all the methods available. A lot of information is provided about the methods; the specification provides their names, what they return, and the type of that return value. This is all the information needed in order to develop programs that use `Date` objects.

Object-oriented programming

You will probably have guessed by now that object-oriented programming is programming using objects. This is a very popular and powerful way to write programs; after all, the reason for the development of C++ from C was to introduce object-oriented features into a language that was already successful and popular. To see why this approach is so popular, let's look at how object-oriented programs are developed.

The first step in writing an object-oriented program is to identify the types of the objects that will be used. This can often be done from a description of the problem area and by discussing the problem with those familiar with it. These object types are specified in terms of their attributes and methods and then programs are written to implement them. Finally, the programs that use the instances of these object types (these instances are called, unsurprisingly, *objects*) are themselves written.

This is the power of object-oriented programming. If programs have been written to implement a particular object type, the code can be reused in many programs that also use the object type. Programmers can quickly build up libraries of working object types that they can use over and over again in many different programs. The savings in terms of time and money can be enormous.



Example 1 – Farmer Strawstrap's farm

Farmer Strawstrap runs a small farm on which he keeps sheep and pigs. He wants to develop some computer programs to help with the running of his farm. What object types will he need?

Farmer Strawstrap wisely plans to identify and then develop the object types to be used in his programs first. He should find that the object types produced can be used in all the programs he develops, and that he will therefore save a lot of time and effort.

The two object types that he needs are fairly obviously *Sheep* and *Pig*. There are many possible attributes, so the farmer would need to think a bit more about the programs that he wants to develop and the information that they will process and produce for him.

The attributes for the two objects would probably be almost the same; this might suggest that one *Animal* object type would be better for this application. A decision on which to use would depend on some further analysis of what the farmer hopes the programs will eventually do; will they have to distinguish between sheep and pigs? The best choice for the object types would also depend on whether he has any plans for developing more programs in the future and, if so, what those plans are.

Some possibilities for the *Sheep* object type are:

Sheep	
Attribute	**Type**
Name	string
Age	integer
Value	Floating point (money)

The *Pig* object type would be very similar, although differences might be uncovered by discussing the farmer's requirements in more detail.

Example 2 – Mollie's library

Mollie has a small library of books that she loans out to the sheep on the farm. Each sheep is allowed to have only one book at any one time. Mollie wants a computer program that will keep details of all her books and will let her keep track of which sheep has which book and when it is due to be returned.

What object types would the program need? What would be their attributes and methods?

Again, it is fairly easy to identify the object types in this example. One will be used for details of the books themselves and one for the borrowers. Even though the borrowers are actually the same as the sheep in Farmer Strawstrap's program the attributes will be different, so it is best to use a different object type. The two could, of course, be implemented as the same object type, *Sheep*. The drawback with this approach is that there are probably very few situations where sheep borrow books and rather more where sheep are sheep. A general-purpose object could be used, but it would involve keeping a lot of attributes and methods that would not be used in the vast majority of the applications that would make use of the object type. At the same time there are many applications where things are borrowed, so a more generic *Borrower* object type is likely to be of much more general use.

This chapter has already looked in passing at some of the attributes that a book object might have. The ISBN attribute of the book object is very useful here; it is guaranteed to be unique for every book. Similarly, it would be a good idea to introduce a "borrower number" for the borrower object type; two sheep might well have the same name, and this will save any possible confusion and embarrassment. The *Borrower* object type might look something like this:

Borrower	
Attribute	**Type**
Borrower number	integer
Name	string
Book borrowed	Book
Date due back	Date

Two of the attributes of this object type are themselves objects. The book that has been borrowed is a *Book* object and the date when it is due back is a *Date* object, as defined earlier in this chapter.

There would probably also be methods for borrowing and returning a book. The return value from the method to return a book could be used to indicate whether the book was overdue when returned:

Method	**Return value**	**Return type**
Borrow book	Nothing	Null
Return book	True if the book is returned in time, false otherwise.	Boolean

Example 3 – Mollie's slightly more complicated library

The sheep like reading but do not like having to make the long walk to return their books. Under pressure, Mollie agrees that each sheep should be allowed to borrow up to four books at any one time. How does this change the object types that have been identified?

When designing object types[4] there is often more than one way. There is more than one possible design and many of these possible designs will be suitable for solving the problem at hand. There are no hard and fast rules; sometimes it just comes down to a matter of the programmer's taste or instinct.

Here is a simple way to extend the *Borrower* class to deal with the new requirement:

Borrower	
Attribute	**Type**
Borrower number	integer
Name	string
First Book Borrowed	Book
First Date Due Back	Date
Second Book Borrowed	Book
Second Date Due Back	Date
Third Book Borrowed	Book
Third Date Due Back	Date
Fourth Book Borrowed	Book
Fourth Date Due Back	Date

There is nothing particularly wrong with this model[5]. It stores the new data well enough and the methods could be extended to record which book had been returned.

An alternative approach involves the definition of a *Borrowing* object type. This is an object that does not correspond to a physical object in the real world but with an event. This is quite common. It would have two attributes: the book borrowed and the date the book is due for return. The *Borrower* object type would simply have as attributes four of these objects.

A final alternative building on this second idea is worth a mention. It would be possible (and this is probably the neatest solution) to define an object type called *Set of Borrowings*. As the name suggests, this would be a collection of *Borrowing* objects. Each borrower object would have one of these collections as an attribute; this is a far neater solution.

The object types in this scheme would be:

Borrower	
Attribute	**Type**
Borrower Number	integer
Name	string
Books Borrowed	Set of Borrowings

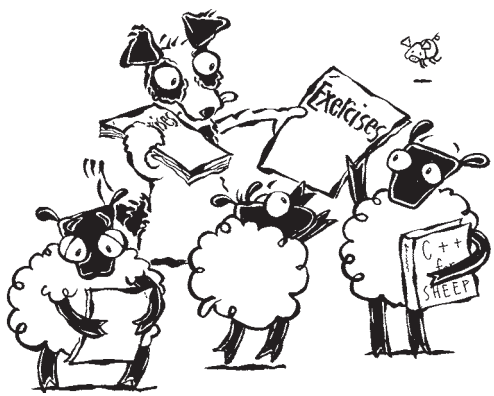
4 And when writing computer programs generally.

5 Well, there is. As we will see later (much later – Chapter 17), this scheme would cause all sorts of unnecessary complications.

Method	Return value	Return type
Borrow book	Nothing	null
Return book	True if the book is returned in time, false otherwise.	Boolean

Borrowing	
Attribute	**Type**
Book Borrowed	Book
Date Due Back	Date

Set of Borrowings	
Attribute	**Type**
First Book Borrowed	Book
First Date Due Back	Date
Second Book Borrowed	Book
Second Date Due Back	Date
Third Book Borrowed	Book
Third Date Due Back	Date
Fourth Book Borrowed	Book
Fourth Date Due Back	Date



4.1 Suggest possible attributes for an object type *Student*.

4.2 Suggest possible attributes for an object type *Professional Footballer*.

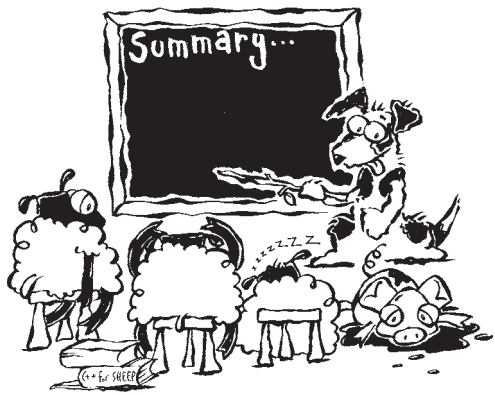
4.3 It is likely that some of the attributes you have suggested for *Student* and *Professional Footballer* are the same. What does this suggest?

4.4 Suggest possible attributes for an object type *Television Programme*.

4.5 Suppose that the *Television Programme* object type were to be used in a program to be used in a video recorder. What methods should be added to the object type?

4.6 A program is to be written to provide details of footballers to a team of match commentators over a season. What methods might be added to the object type you have already defined?

4.7 Mollie plans to allow the sheep to reserve a book that has been borrowed by another sheep. What should be changed in the object types identified in the third example?



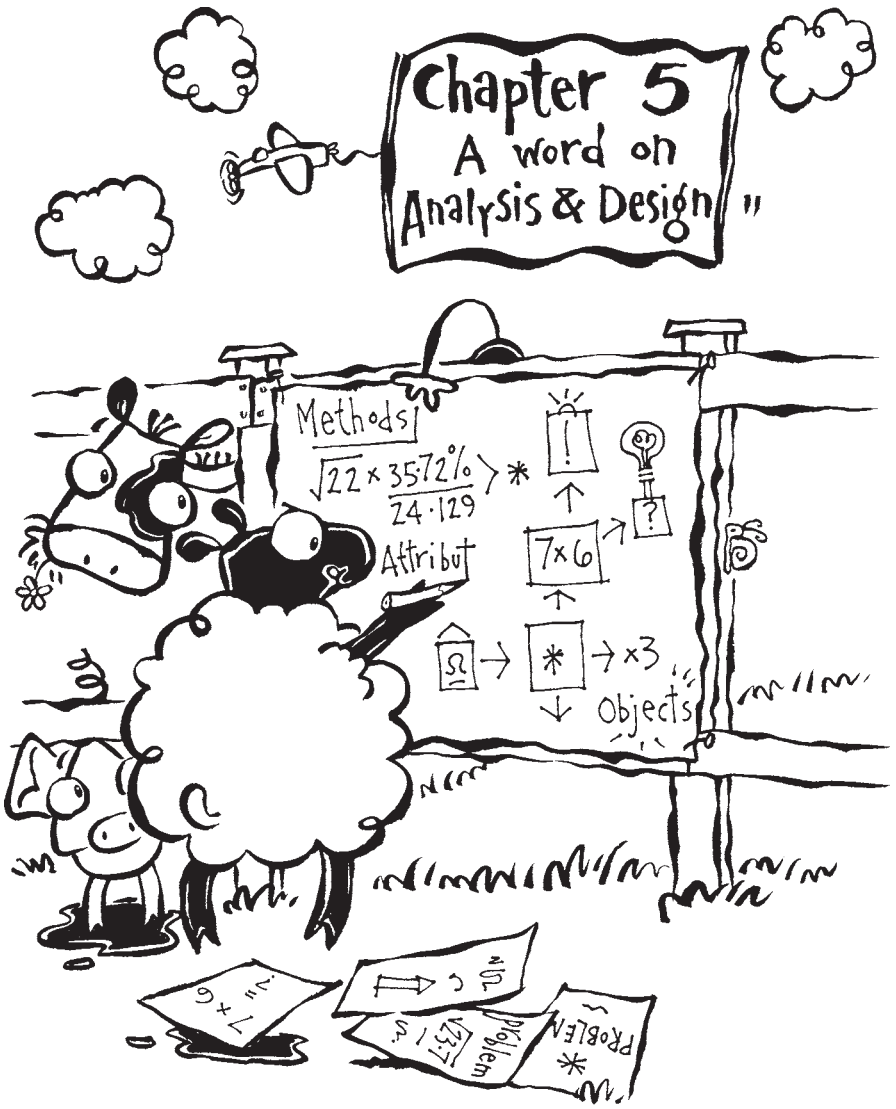
C++ is an object-oriented programming language. C++ programs define and process objects that represent physical objects or events in some real world problem. The first stage in developing a C++ program is to identify the object types in the problem and to write programs to implement these. These object types are then used in the programs that are written to solve the problem.

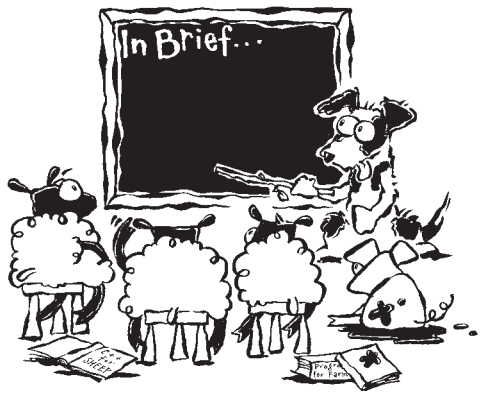
An object type is used to represent a group of connected objects in a real-world problem. Individual instances of this object type (objects) are characterised, usually uniquely, by the values of their attributes. Attributes are values of interest; they have a type and can store only one value. Attributes may themselves be objects.

As well as attributes, object types also have *methods*. A method is a procedure that processes the attributes of an instance of the object type to produce some effect. A method may or may not change the values of the object's attributes.

A key principle of object-oriented programming is code reuse. When an object type has been written for one problem area it can be reused in another, ideally with no changes. This has the potential to save a great deal of time and money.

You should now understand what an object is. It's time to move on to look in a little more detail at how the objects in a problem area can be identified.

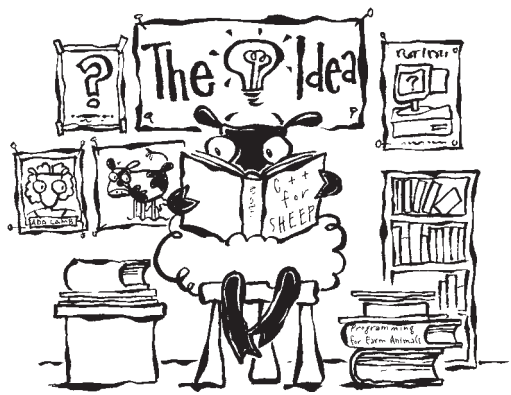

Chapter 5
A word on
Analysis & Design
Methods
Attribut
Objects
PROBLEM
Problem



The exercises in the previous chapter introduced you informally to the task of identifying objects in a real-world problem. You should now have some experience of suggesting the objects that might be used in a program to solve some problem; you should also be able to make some suggestions for possible attributes and methods of the objects you identify.

This chapter takes this a little further to suggest a framework that will allow you to carry out this important task in a structured way. Before this book moves on to describe features of C++, it is important that you realise the crucial importance of good program design. You must understand that, before any program is written, the problem area should be carefully investigated and analysed, and the solution carefully designed.

After reading this chapter you should have a basic idea of how to identify possible objects and object types in a problem area. You should be able to take a description of a problem and use it to create a list of object types, together with their attributes and methods. You should be able to present this in a way that would be useful to a programmer setting out to implement your solution.



A computer program is written to solve a particular problem or to carry out a particular task. Before this program can be written there are two important tasks that must be performed. First, the problem must be carefully analysed, and then the solution must be designed. Sometimes more than one solution will be designed to allow the programmers a choice. The activity of analysing

a problem and designing a solution is normally called "Systems Analysis", and those that specialise in this task are called "Systems Analysts" or just "Analysts". It is common in today's IT industry for the programmer and analyst to be the same person (an "Analyst Programmer"), but this is not always the case.

This is not a book about systems analysis, so this chapter gives you only a very brief insight into the area. At this stage you need to know enough about analysis and design to allow you to make sensible choices as you write your own programs. The activities of analysis, design, and programming are clearly closely linked. You may well find out that you enjoy the analysis or design aspects of developing programs more than the actual programming; that's part of what learning to program is all about!

Before you start on this chapter it is important that you understand that the activities described in this chapter are normally best carried out a long way away from a computer. Analysis involves studying a problem and talking to the people affected by it. Design is a pencil and paper exercise, with a computer used, if at all, to do little more than help with drawing diagrams or preparing documents. When you write programs you should never sit down in front of a computer without a design at your side. You should have a clear idea of what your program is going to do and how it is going to do that; this idea is the main output of thorough analysis and design.

There are several stages to analysis and design. These are often described in the form of a structured *methodology*, a formal way to carry out the various tasks. There are many formal methodologies that can be used for analysis, and you might well meet one if you study systems analysis further. For simplicity this chapter does not describe a particular methodology; instead, you will find a suggested set of steps to take. This suggestion should be particularly suitable for new programmers.

Identify object types	The first step is to identify the object types in the problem area. In the spirit of code reuse it might even be the case that some of these object types have been met before in previous programs, so working implementations for them may well already be available.
Identify attributes	The attributes that describe each object type are identified and added to the definition. The type of each attribute is also identified; this might lead to the identification of more object types that are found to be attributes.
Identify methods	The methods are added.
Design methods	The methods are effectively each small programs, so each one is designed separately.
Design program	Finally the complete program that will make use of the object types identified can be designed. Only when this step is complete is the program actually written.

These steps will be illustrated by means of an example. Ready?

Identifying object types

This is the most fundamental step. The object types will be the basic building blocks of the final program, and it is crucial to identify the correct ones. A simple way to identify candidates for objects is to take a description of the

problem area and to extract all the nouns (or noun phrases); these are the words that refer to "things" that exist in the problem. Such a description would normally be a result of an interview with someone close to the problem.

To illustrate this, here is a description of a problem:

> *The farm is situated in North Yorkshire. Farmer Strawstrap owns the farm and together with his family takes care of all the animals. There are many sheep, many pigs and two horses on the farm. The farmer also has a dog, called Mollie. A program is required that will assist with the administration of the farm. The farmer wants to be able to keep track of all of his animals; he needs to know the name, age and approximate value of each. When he sells an animal he needs to be able to record this fact together with the name of the farmer he has sold the animal to. He never sells his horses or his dog.*

It's easy to go through this description marking the nouns (here underlined):

> *The farm is situated in North Yorkshire. Farmer Strawstrap owns the farm and together with his family takes care of all the animals. There are many sheep, many pigs and two horses on the farm. The farmer also has a dog, called Mollie. A program is required that will assist with the administration of the farm. The farmer wants to be able to keep track of all of his animals; he needs to know the name, age and approximate value of each. When he sells an animal he needs to be able to record this fact together with the name of the farmer he has sold the animal to. He never sells his horses or his dog.*

This gives a list of things that might be objects in the final program.

Some of the nouns in the description are actually different names for the same thing, so the next step is to remove synonyms to give a shorter list. At the same time, some of the words in the description don't quite describe the potential object fully, so the names should be refined to make them clearer:

- Farmer Strawstrap's farm
- North Yorkshire
- Farmer Strawstrap
- Farmer Strawstrap's family
- animals
- sheep
- pigs
- horses
- dog
- Mollie
- program
- administration
- name
- age
- approximate value
- name (of farmer buying an animal)
- farmer buying an animal

Some of these can be eliminated at once. "program" and "administration", for example, are clearly nothing to do with object types. Proper nouns can also usually be eliminated; they refer to a particular instance of an object type that is very probably already covered elsewhere. Finally, some nouns will actually be attributes of other objects; here "name", "age" and "approximate value" clearly fall into this category. The list is now much shorter:

- Farmer Strawstrap's farm
- Farmer Strawstrap
- Farmer Strawstrap's family
- animals
- sheep
- pigs
- horses
- dog
- farmer buying an animal

Information about the tasks to be carried out by the program will enable the list to be trimmed further. "Farmer Strawstrap's family" is only mentioned in passing, so that can be removed. Much the same goes for the farmer himself and the farm. "animals" is a general term that covers the sheep, pigs, horses and dog; it can also be removed[1]. This gives a final shortlist:

- sheep
- pigs
- horses
- dog
- farmer buying an animal

The description also provides the information that the farmer would never sell his horses or dog. Given the nature of the program it might be reasonable to leave them out, but then again the farmer might change his mind in the future. Part of the analysis process would be to go back to the farmer and ask for more details and a more detailed description of the problem. This would allow a sensible and informed decision to be taken.

For the moment the assumption will be that neither is required; this seems reasonable on the basis of the limited information currently available. The list becomes:

- sheep
- pigs
- farmer buying an animal

The final step is to choose the names for the object types. Singular names are customary, so pigs becomes "pig" while sheep stays the same (!). The

1 This actually shows, of course, that there is an object type "animal" in this problem and that it has four separate object types that represent particular types of animal. If we were to find out that each type of animal had the same attributes, it might well be possible to implement a single "animal" object type that would contain an attribute to indicate the type of animal.

current name for the object type representing the farmers that buy animals is rather unwieldy; this could be changed to simply "farmer". The final list of object types in this problem area is then:

- sheep
- pig
- farmer

Identifying attributes

Finding the objects has provided a useful head start in finding some of the attributes. A check through the discarded candidates for objects reveals "name", "age" and "approximate value", all of which are attributes of the *Sheep* and *Pig* object types. It's likely that they are also attributes of the other two animal object types (horse and dog) as well, but this would only be relevant if these were to be implemented.

There is less to go on in the description for the attributes of the object type that will represent the farmers who buy the animals. In practice an interview would again be used to determine these, but a reasonable guess for them would be to include attributes such as "name", "address" and "telephone number".

Finally, there is a requirement to record the identity of the farmer to whom an animal was sold. This can be introduced in two ways; either an attribute "buyer" can be added to the *Sheep* and *Pig* object types, or an attribute to represent the animals bought by each farmer could be added to the farmer object. There is no good reason at this stage to believe that either of these possibilities is better than the other; the choice could be left to the final programmer or might be determined when some more details of the requirements of the final program are known.

The types of most of the attributes are straightforward to establish. The only one that might present a choice is the telephone number of the farmer. It might seem at first sight as if this should be an integer, but a string is probably to be preferred. Telephone numbers are traditionally written with spaces to make them more memorable, and often begin with a 0; handling these values sensibly would be very difficult to achieve if the number were stored as an integer. It is also unlikely that the final program would need to perform arithmetic on a telephone number (an operation on integers), but it is possible that the program might need to extract certain digits from the number (an operation on strings). A string is best.

Finally, the names of some of the attributes could do with some simplification; in particular, "approximate value" should be simply "value". This gives the final set:

Object type	Attribute	Type
Sheep	name	string
	age	integer
	value	floating-point
	sold to	Farmer
Pig	name	string
	age	integer
	value	floating-point
	sold to	Farmer
Farmer	name	string
	address	string
	telephone number	string

Identifying methods

The task of identifying the methods of the object types involves returning to the description of the problem area. This time the focus is on the purpose of the final program; it is vital that the object types have sufficient and correct methods to support this. In particular, the analyst should be looking for any *transactions* that will change the values of any attributes of any instance of an object type. Transactions often show up as events in the problem description.

In this example there is only one obvious candidate: the farmer sells an animal. This transaction would change the value of the "sold to" attribute of the object instance representing the appropriate animal; it is therefore a method of both object types representing animals, *Sheep* and *Pig*.

The description also contains the information that the farmer "needs to know the name, age and approximate value" of each of his animals. This suggests two things; that methods will be needed to display this information to him in some neat format and that methods will be needed to change this information. It seems reasonable to assume that the name of a sheep never changes, so methods will be needed only for age and value.

The *Farmer* object type has no obvious methods. The only possibilities seem to be methods to change the values of the attributes, but there is no evidence that this would be a common requirement. This is another example of something that would have to be confirmed with a further interview.

Of course, Farmer Strawstrap will also want to use the information about his animals for many other purposes. He might want to display lists of all his animals or find all the animals of a certain value. These are not, however, methods of the object type; they operate on collections of objects and not on individual objects. These are functions that would be carried out by programs that use the objects and they should be left to the programmer to implement.

This analysis gives the following collection of methods for the *Sheep* object type. The methods for the *Pig* object type are the same as those for *Sheep*, and the *Farmer* type has no methods.

Object type	Method	Description	Returns	Return type
Sheep	display	Displays details of a sheep in a neat format	Nothing, just display on screen	Null
	sell	Records that a sheep has been sold and stores the name of the farmer buying	Nothing, underlying values are changed	Null
	changeAge	Changes the sheep's age	Nothing, underlying values are changed	Null
	changeValue	Changes the value of a sheep	Nothing, underlying values are changed	Null

Designing the methods

Each method must now be designed. This involves specifying the effect of each method in detail. It is normal to use a structured format for writing this, involving the specification of:

- the purpose of the method
- the inputs of the method
- the values that are changed by the method
- any side effects (such as output on a screen or changes to a file of data) that the method may produce

The format of this information should be fixed, and in practice might well be recorded on some suitable form.

For example, the specification for the *display* method of the *Sheep* object type would be:

```
Name:           display
Object type:    Sheep
Purpose:        Displays the details of a sheep on the screen
Inputs:         a Sheep object
Values changed: None
Side effects:   Display on the screen
```

This specification would be extended in the final design process to include a precise description of the layout for the display; this might well make use of a diagram.

The *sell* method is more complicated; the programmer must provide the identity of the farmer to whom the sheep has been sold. This method also carries out a rather more complicated task than simply displaying the values on the screen, so a more lengthy description of its effect would be provided:

```
Name            sell
Object type:    Sheep
Purpose:        Records that a sheep has been sold.
Inputs:         a Sheep object and a Farmer object
Values changed: "sold to" attribute of the sheep
Side effects:   None
Description:    The value of the "sold to" attribute of the sheep
                should be changed to store the Farmer object
                provided. If this value already indicates that the
```

```
    sheep has been sold an error message should be
    displayed and no changes should be made.
```

Obviously this detailed description can become very complex. Worse, it can be very hard to write it with precision; if the programmer misinterprets what the analyst has written the final program will not work as expected.

For this reason it is useful to be able to write the description out in a slightly more formal and unambiguous way. The normal way to do this is to write in a form of *pseudo-code*. This is a form of language that is much more terse and precise and is, in fact, quite close to a computer programming language. There are no hard and fast rules for writing pseudo-code (although some software companies may have a preferred or recommended style), except that anything written should be clear and unambiguous. The description of the processing carried out by the `sell` method could be written in pseudo-code as:

```
IF the value of "sold to" is blank THEN
  Set the value of "sold to" to the Farmer object
OTHERWISE
  Display an error message - the sheep is already sold
END IF
```

Armed with the pseudo-code, the programmer will be in no doubt about the analyst's intentions. It is good practice to write out all but the very simplest methods in this way before writing them in actual program code.

Designing the program

With the object types designed, the program or programs that will use them can also be designed. As with the design of the methods, once the purpose of the program has been specified it is usual to describe it more formally using pseudo-code.

One program that is clearly needed is one that will allow Farmer Strawstrap to record the sale of a sheep. This would presumably ask him to enter the name of the sheep and the name of the farmer buying, and would then make the required changes in his records. The pseudo-code for this might be:

```
Display "Enter the name of the sheep: "
Read Input and store as Sheep
Display "Enter the name of the farmer: "
Read Input and store as Farmer

Use sell method of sheep object to record sale
```

This simply and unambiguously specifies the method and tells the programmer precisely which other method to use.

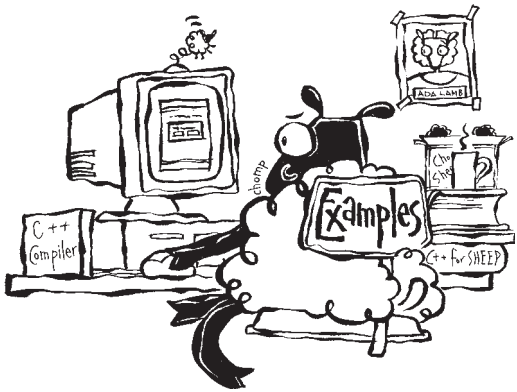
As another example, suppose that Farmer Strawstrap decides that he wants a program that will display the details of all his pigs. This program will have to process and display the details of each pig in turn, using the `display` method. This could be expressed in pseudo-code:

```
Get the details of the first pig
WHILE there are pigs left
  Display the details of the pig using the display method
  Get the details of the next pig
END WHILE
```

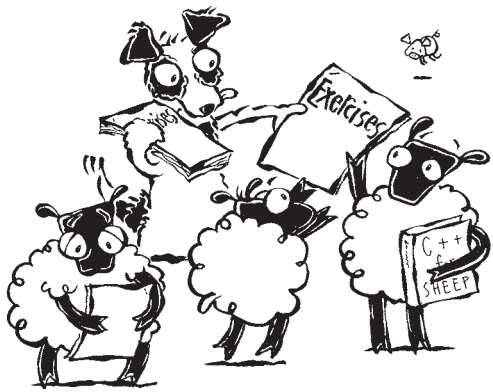
The design of anything more than simple programs will have to wait until you have learned more of the details of programming. For the moment you should just appreciate the importance of design and the great importance of expressing the design in an unambiguous way.

Analysis and design

This example has shown how a description of a problem can form the basis of an analysis of a problem area. This analysis leads to a design of a solution. The approach suggested here is a simple methodology that you should follow whenever you come to develop a new program.



Since this chapter included one long example there are no extra examples this time. On with the exercises!



5.1 Identify the object types that would be needed in a solution to the following problem. For each object type you identify, suggest attributes and methods.

The sheep on Farmer Strawstrap's farm have taken up cricket. They have formed themselves into four teams and are organising a knockout competition. They have asked Mollie and Peter to be umpires. Flossy has been asked to write a program to record which sheep is in which team and to keep a record of the results of the matches.

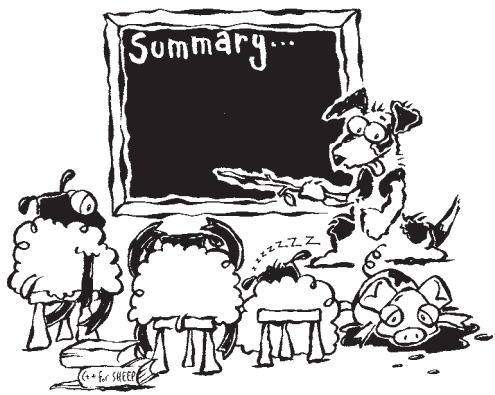
5.2 After the success of their first competition the sheep decide to organise a league. Each team plays each other and two points are awarded for a win, one for a draw and none for a defeat. Flossy is asked to write a new program. How does this change the design of the object types?

5.3 Flossy has a file containing the results of all the matches in the league. Use pseudo-code to design a program that would process the file to discover how many points each team has got so far. You can read through the file many times.

5.4 Express the following description of a program in pseudo-code. Assume that the program uses a *Book* object type that has an attribute "borrowed by" storing the sheep that had borrowed the book.

> *When a book is returned Mollie enters the name of the sheep and the title of the book. The program finds the information about the book and records the fact that it is back in the library.*

5.5 Examine the job advertisements in a computing trade newspaper such as *Computing* or *Computer Weekly*. Note down the job titles for all the jobs relating to programming. How many jobs are offered as Programmers? As Systems Analysts? As Analyst Programmers? What salaries does each job seem to attract? What are the different skills or experience requirements for each type of job?

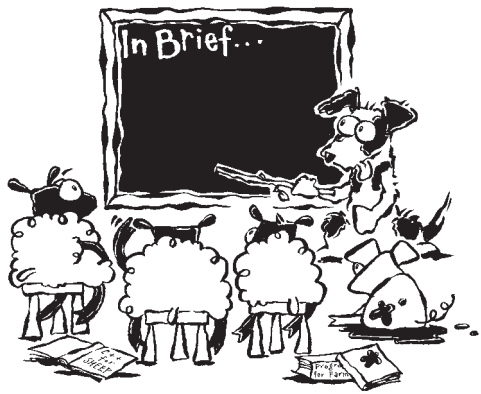


Before any program can be written the problem that it is to solve must be thoroughly analysed, as must the area in which the problem exists. This analysis, which is often carried out by a Systems Analyst, forms for the basis of the design of a solution. The programmer works with this design to produce the final working program.

The first stage in the analysis is the identification of the object types that will be used in the program. Their attributes and methods are also found and the methods are designed. This design normally makes use of pseudo-code to ensure that it is unambiguous. The ambiguity that can result from the use of normal natural written language is best avoided. The use of pseudo-code is also usual and useful when designing programs.

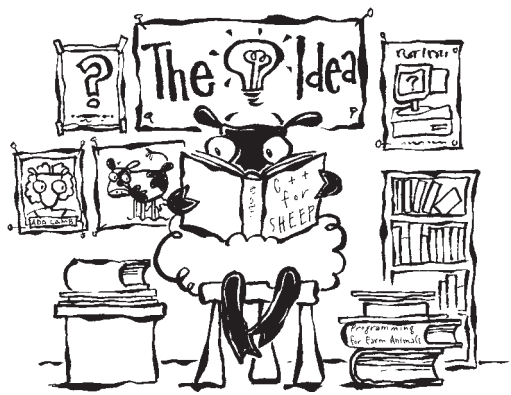
Thorough analysis and design are essential before programming can start. Now you're getting closer to doing some actual programming!


Chapter 6
Programming
(Don't Panic!)
YOGA
for sheep
STYLE
Indentation



The time has come when you are almost ready to start using C++ to write some programs. Before you do, though, you need to understand something about the process of programming. Programming is a highly structured process; some might even call it a discipline. Good programming requires that the programmer adopt a sensible, structured and measured approach. Above all, it requires that the programmer doesn't panic!

After reading this chapter you should understand why a structured approach is essential when writing programs. You should appreciate the importance of always applying good consistent style and layout when you come to write your own programs. You will have met the five golden rules of good programming practice and style. Finally, after a long wait, you will also have seen your first C++ program.



Suppose that you were about to build a new house in the middle of a large empty field. This is one strategy that you might use. You could make a list of all the things that you would need; you would include bricks, cement, wood, paint, windows and so on. Then you would buy them and arrange them in a huge random pile in the middle of the field. Over the next few days you would poke around at the pile and sort them out and arrange them into a house.

This strategy might just work, but many things might go wrong. You might put the bricks on top of the windows and break them, or you might lose the

paint under a large pile of wood. The risks of disaster are so great that in practice you wouldn't seriously consider using an approach like this.

Suppose that your car was broken and you took it to the garage for repairs. You would expect the mechanic to listen to your description of the fault and then spend some time diagnosing what exactly was wrong. When the fault had been identified you would expect a quick and easy repair. You would certainly not expect the mechanic to spend time randomly replacing parts of your car until it worked again. This random approach would probably work, but it's far from the cheapest or most efficient way of repairing a car.

These two examples highlight tasks where a structured and methodical approach is essential. You will not be surprised to learn that programming is another such task. There is a set of steps that must be followed, in order, when any program is written. Failure to follow these steps is a sure recipe for disaster. Much the same applies to following the steps in the wrong order or jumping randomly between them.

Writing a program

A user who requires a new program will expect a programmer to go through a process that is very similar to that which you would expect from a car mechanic. The first step is to carry out an analysis to acquire a thorough understanding of the problem. This is followed with a design of the most appropriate solution, which is then quickly and efficiently implemented as object types and programs. It would be a huge mistake to start writing the program before the problem had been fully investigated and understood. It would be equally foolish to start writing the program before it had been properly designed.

The process of writing the program itself follows a similar pattern. The first stage is to implement the object types that will be required. These are tested fully in isolation so that the programmer is sure that they work correctly. Only when they have been thoroughly tested should they ever be used in a program. Once again it is extremely foolish to start writing a program that uses some object types before the object types have themselves been written and tested. Such a mistake would only lead to a great deal of extra and unnecessary work as more and more errors had to be located and corrected. And worse, if a program is written before the object types have been fully tested it is very difficult to know for certain if the errors are in the program or in the object types themselves.

Even the production of the program code itself follows the same pattern. The code should always be developed in a steady incremental manner. Functionality should be added to the program very gradually, and any new function should be tested and shown to work before any further development is attempted.

At some time in the future you will be writing a program and you will not be able to get it to work. You will have the chance here to make a huge mistake. You will have the chance to ignore this problem and then start to work on some other part of the problem. If you do this you are only making worse trouble for yourself! Never start working on a different part of the program when you find that you can't go any further in one area; find some help and sort out the first problem.

The first golden rule is:

Work on one function at a time. Get it right and then go on to the next one. Never work on many functions at once!

As you learn to program you will probably get the chance to see some experienced programmers at work. One thing that you will notice is that they have to deal with far fewer errors from the compiler than you seem to. This is only natural; you are bound to make more mistakes as you learn. At the same time, though, the experienced programmers are doing some things that will make sure that they have as few errors as possible to find and correct.

An experienced programmer will compile the program many times before it is complete. Sometimes this will just be to check that the syntax of the program so far is correct, and other times this will be to quickly test some new function. An experienced programmer will never, ever, type in a complete program and then start to force it to compile. If you choose to adopt an approach like that you have made a very big mistake; you have chosen the best way to make sure that you waste time and get practice in correcting as many errors as possible. It is in the nature of programming that a simple error at the top of a program file (something as insignificant as a single semi-colon) can cause errors throughout the program. If the program is several hundred lines long this will be several hundred errors, but if the program is only ten lines long there will be far fewer. It is always best to find such errors in very small programs, before their effects become so big as to obscure the original cause.

You will come to need to compile your program just to check the syntax less and less as you gain more experience. As you are just starting out you should remember to compile your program, if only just to test the syntax, every time you add around twenty lines to it.

The second golden rule is:

If you have added twenty lines to your program, finish the part you are working on, compile it and check that your syntax is correct. Fix any errors before you carry on.

Style

A particular aspect of programming is a programmer's style. Every programmer has their own style in writing programs, something that is probably unique to each individual. It's much the same thing as handwriting; you can probably recognise your own handwriting even if you wrote something many years ago. A programmer can probably recognise a program that they wrote many years ago.

Any programmer would hope to be able to take a program that they had written many years previously and make changes to it. Many programs remain in use for a very long time and changes often have to be made many years after the program was first written. Of course, sometimes the original programmer will not be available to make the changes. In this case a new programmer must be able to take the program, quickly understand it and then make the required changes. This is much more complicated to achieve if the program is written in a poor style. Poor style can make even the most useful program totally impenetrable to even the best programmer. Good style is an essential feature of good programming.

As you learn more and more you will come to develop your own style. Like handwriting, style is very much a matter of taste; you will start to think that

certain parts of a program look better to you if laid out in a certain way and you will start to adopt that style. All the programs in this book are written in the style that I prefer. I'm not suggesting that this is a particularly good or bad style, but it's the one that I have developed over many years and is the one that I am happiest with. You are free to copy it as you learn, but you will probably eventually adapt it as you develop your own set of preferences.

There is no such thing as a good or bad style, as long as the style follows some basic rules. It is essential that any style is consistent. If a programmer chooses to use a particular way of laying out a particular part of a program the programmer should use precisely the same layout every time they write a similar piece of program. As a first example, here is a very simple C++ statement:

```
number = 1;
```

That line is laid out as I would do it if I were I writing a program. There is a space before and after the equals sign. Other programmers, including many of those that I have worked with, might write it like this:

```
number=1;
```

with no spaces, or even:

```
number = 1 ;
```

with a space before the final semi-colon. None of these styles is better than the others; the programming statement is perfectly clear in each case. What is crucial is that a programmer choosing any of these three styles should use the same layout in every similar statement. It should be obvious that a program that contained a section like this:

```
number = 1;
anotherNumber=2;
aThirdNumber = 3 ;
```

is an example of very poor and inconsistent style.

For the moment it probably does make sense for you to follow the style of the programs in this book. The only exception to this would be if you are required to follow another for some reason; perhaps the course you are following expects you to use a particular style. Whatever you do, make sure that the style you follow is consistent.

The third golden rule is:

Whatever style you use, make sure that it is clear and neat. Remember that someone may later have to change your program without your advice.

As well as being consistent, a programming style needs to be easy to read. In C++, the example statement above could be written:

```
number
=
1
;
```

or even:

```
number          =            1          ;
```

Hopefully it is obvious why these two are not good examples of style! Once again you are free to follow the style of the examples in the book, or you are welcome to develop and adopt the style to something that you prefer.

Sometimes software companies will require their programmers to write programs using a particular style, often called a "house style" and enshrined in a hefty document labelled "Coding Standards". The reasons for this are sound; it ensures that all programs written in the company are consistent in style and so increases the chances that programs will be quickly understood by other programmers. Many programmers do not like working to such restrictive guidelines; programming is an essentially creative process and such strict controls reduce the chances for creativity. It is possible that you are required to write your programs to conform to some house style; if you do then you will have to adapt the programs in this book to fit the style.

The fourth golden rule is:

As you program, adopt and apply a consistent style. As you learn more, develop this style to something that you feel confident and happy with.

Indentation and layout

A particularly important aspect of a program's style is the way that it is laid out. In the same way that a page of a book or magazine can be laid out well or poorly laid out, so can a program. A program's layout can add a great deal of meaning to a human reader. The layout is quite irrelevant to the compiler, which will simply ignore it, but it can be extremely useful to a human trying to understand the program.

Layout can emphasise which parts of the program serve a connected purpose; for example, blank lines might be left in a program to show the connection between certain lines or something about the program's structure. A crucial aspect of layout is *indentation*, where spaces are left at the start of lines. The indentation of a program can show a reader a great deal about the program's structure.

As an example, here at last is your first C++ program! At the moment you won't understand much of what it does, but that's not the point. Just concentrate for now on the way that it's laid out. All you really need to know at the moment is that the lines that start // are just intended for the human reader; they are *comments* and are completely ignored by the compiler. Comments are used to explain how a program works; you should find that by reading them you are able to see what this program does even if you don't understand any of the C++.

```
// layout.cc
//
// Program to show some aspects of layout.
//
// Chapter 6.
//
// Author:    AMJ
// Date:      20th November 2001
// Platform:  g++, Linux

#include <iostream>
#include <string>

using namespace std;
```

```
int main ()
{

  // Variables - one to store name and one for age

  string name;
  int age;

  // Input - get name and age

  cout << "Enter your name: ";
  cin >> name;
  cout << "And enter your age: ";
  cin >> age;

  // Process - if name is Tony, set age to 21
  // otherwise add ten years

  if (name == "Tony") {
    age = 21;
  }
  else {
    age += 10;
  }

  // Output the name and age

  cout << "Hello, " << name
       << ". It interesting that you are "
       << age << "." << endl;

  // Exit the program

  return 0;

}
```

I hope you will agree that this program is quite neat. There are blank lines separating the parts of the programs that carry out different functions and generally separating the various lines of the program. The indentation is also important; the { and } (called *braces*) mark the start and end of the program so all the statements inside the program are indented by two spaces to emphasise this. One statement is controlled by the "`if`" statement (see the braces again), so it is also indented.

Now for an example of poor layout. Believe it or not, this program is effectively identical to the previous example. The compiler would treat it exactly the same and it would behave identically when run.

```
#include <iostream>
#include <string>
using namespace std;int main (){string f;int b;
cout<<"Enter your name: ";cin>>f;cout<<"And enter your age: ";
cin>>b;if(f=="Tony"){b=21;}else{b+=10;}cout<<"Hello, "<<f
<<". It interesting that you are "<<b<<"."<< endl;return 0;}
```

It is obvious that this program is much harder to understand. There are no convenient blank lines to break it up into small chunks and there is no indentation to help with an understanding of the structure.

Another point about layout is that all lines in a program should be kept reasonably short. The purpose of this is to make sure that the program will fit neatly onto a printed page. It is customary to make sure that no line in a program extends longer than 80 characters; the number 80 dates from the time when this was the maximum number of characters that could be displayed on one line of a terminal or printed across a page. These days this practice remains a good one; most experienced programmers will always make sure that their lines are kept short, and many would stop at around 75 characters to be on the safe side. All the programs in this book have been formatted to make sure that all lines fit neatly on the page (sometimes this means that the lines are rather shorter than 75–80 characters).

When you write your programs always remember that you are not just writing them for yourself. You are also writing them for other programmers, and these are programmers who may one day have to change your program. The indentation and layout that you are going to use are of enormous help to others reading your program. Often when students ask me to find an error in their programs they seem to be amazed when the first thing I do is send them away to go through and correct their indentation; often this alone finds the error, but even if not it always helps.

Always write your programs with good layout and indentation. Never be tempted to write your programs with no indentation, intending to indent it all properly once the program works. This is simply something that will make more work for you and will make the job of anyone that helps you a great deal more difficult. In many ways it is as important that a program is neatly and consistently laid out as it is that the program works correctly.

You may well find that the C++ system that you will be using will lay out your code neatly for you as you program; consult your *Local guide* to see if you have this facility. If it will do, you should check the style that it will follow and you should decide if you are happy with the style. If you are not, spend some time finding out how to customise the style to your own preferences.

The fifth and final golden rule is:

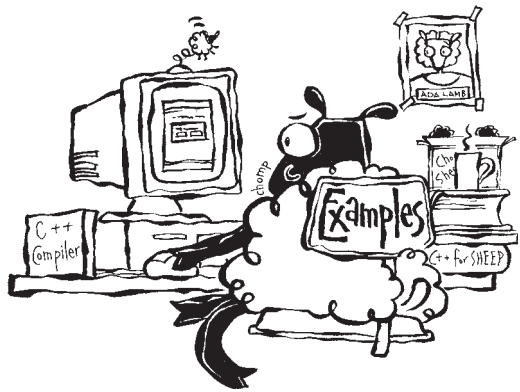
Always pay attention to layout and indentation as you develop your program, and never treat either as an afterthought.

Don't panic!

You will realise by now that programming is a complicated business. As well as producing a program that correctly carries out some useful task, a programmer must always produce a program that is neatly and consistently laid out. A program is only ever of any use if it can later be changed and adapted by other programmers.

Effective programming is all about applying a set of rules to a problem. When you become confident and proficient you will probably find that programming is an extremely rewarding task. A fully working program is a fine creation and something to be proud of. Programming itself is an extremely creative and enjoyable process. With neat layout and good style a program is very nearly a thing of beauty.

As you learn to program, remember that it is a structured process. It is a process that can only be carried out by someone who is calm and collected. As you learn, above all "Don't panic!".



Example 1 – Freda's poor style

As a new C++ programmer, Freda is very concerned that her programs work. Mollie has set her a lot of exercises and Freda has to work hard to complete them before the deadline. She decides to ignore the indentation of her program and intends to correct it all when the program works. Unfortunately, it takes a long time to make the program work and Freda doesn't have time to correct the indentation. She is somewhat dismayed when Mollie refuses to mark her program.

Why does Mollie behave like this? Is this reasonable?

This is very reasonable. Indentation is a crucial part of programming style and it is important that new programmers quickly get into the habit of producing correctly indented and well laid out programs. Freda must remember that her programs may well be used and corrected by other programmers in the future and her poor layout will make it very difficult for these programmers to discover how her program works. Mollie is quite correct to refuse to mark the program; the poor style will make it very difficult for her to see how it works or to spot any errors.

Freda should realise that it is just as easy to create the program with proper indentation as it is not to bother. It is probably even easier, as the indentation can often help highlight errors before the program is even compiled. Leaving the indentation to the last minute is a very poor strategy.

Do not make the same mistake. Make sure that your programs are correctly and consistently indented as you write them. As you learn more C++, adapt your style and develop one that you can confidently apply consistently. Whatever you do, do not ignore style and do not treat it as an incidental extra that can be added once the program works.

Example 2 – Peter's poor style

Peter is enjoying learning C++ but he finds typing difficult. So that he can learn more C++ quickly, he decides that he won't bother with any comments in his programs. He too is dismayed when Mollie refuses to mark his comment-less programs. Is Mollie right?

Mollie is quite correct again. Like indentation, comments should never be treated as an afterthought. They should be included in programs as the program is written to highlight the various stages that the program is passing through. Comments are crucial when another programmer comes to correct or maintain any program. It is always easier to add comments as the program is

written; if they are added at the end, after a long period of development, the original programmer will often find that they themselves have forgotten how the program works!

Again, do not make the same mistake. Get into the habit of adding comments to your programs as you write them.

Example 3 – Flossy's comments

Flossy is alarmed at Peter's comment predicament and decides to add comments throughout her program. She adds comments to each line; many of the comments are quite long and eventually there are more comments than functional lines of program. Flossy is as dismayed as the others when Mollie says that her program is no better. What is going on?

You will not be surprised to learn that Mollie is quite right. A programmer who includes too many comments in a program is almost as bad as a programmer who includes none at all. A program with too many comments quickly becomes cluttered and very difficult to read. The key to good style is that the program is neat and easy to read and understand. Too many comments, and comments on lines whose function is obvious, can get in the way.

You may well find that as you become more and more confident in C++ you will include fewer and fewer comments in your programs. You may be tempted at first to comment almost every line, and this is fine. As you learn more you will come to realise that this is not really necessary as there are some lines that every programmer will understand. Don't be afraid to let your style evolve and adapt.



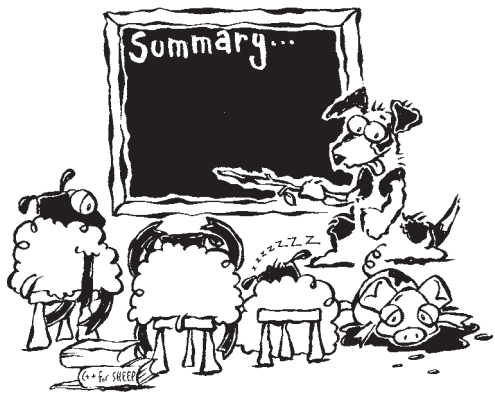
6.1 Find another C++ book and look at some of the example programs. See if you can pinpoint the differences between the style in that book and in this.

6.2 Find out if you are required for some reason to use a particular style as you learn to write C++. If you are, find out what the differences are between the style that you will use and the style used in this book.

6.3 Investigate any facilities that your C++ system might have for automatically formatting your C++ programs. Do you need to customise any of these features?

6.4 If you can, get hold of C++ programs written by two or more different experienced programmers. What common features can you see in these two

styles? Do you prefer one to the other? Even though you are still a novice programmer, how much of the programs can you follow? Does the style help?



Programming is a structured task that requires a structured approach. A programmer must build on a thorough understanding of a problem to produce a program; a program must never be developed before the problem that it is intended to solve has been fully analysed. A programmer must follow this process strictly.

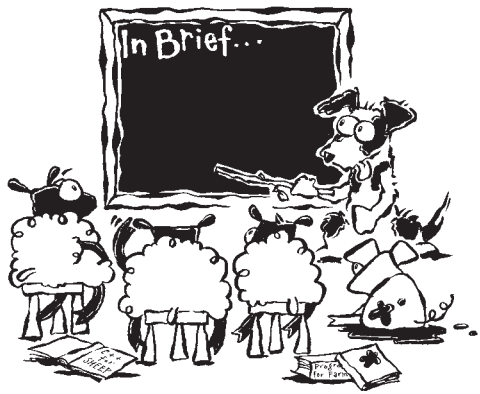
Good and consistent programming style is as important as a sensible structured approach. The guidelines for good style can be summarised in five golden rules:

1. *Work on one function at a time. Get it right and then go to the next one. Never work on many functions at once!*
2. *If you've added twenty lines to your program, compile it and check that your syntax is correct. Fix any errors before you carry on.*
3. *Whatever style you use, make sure that it is clear and neat. Remember that someone may have to change your program without your advice.*
4. *As you program, adopt and apply a consistent style. As you learn more, develop this style to something that you feel confident and happy with.*
5. *Always pay attention to layout and indentation as you program, and never treat either as an afterthought.*

Remember these rules as you learn to program. If you break them you will be making more and more trouble for yourself.

Now, finally, the time is right for you to meet some C++ and to write your first program.

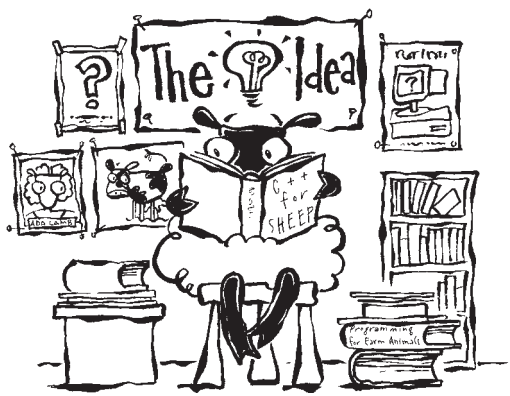

Chapter 7
The Basics
. . . 8»
int
Flossy
string
float
true
bool
char



This chapter is the first one that looks in detail at some C++. You should by now understand what a program is and you should have some idea of how to analyse a problem area and design a solution that will solve a particular problem. You should also know how to create, compile and execute the programs you write. Now it's time to actually write some!

A C++ program is basically nothing more than a set of *statements* that instruct a computer to perform some task. At the simplest level a program takes some values and processes them to produce some other values. In this chapter you will meet a full C++ program in detail for the first time and will see how it stores and processes these values. The next chapter will go on to explain how to display the values and how to ask a user to enter the values. Together these two chapters cover the most basic tasks that can be carried out by a C++ program.

After reading this chapter you should understand the basic structure of a C++ program. You should understand how values can be stored in *variables* and how they can be manipulated. You should then be able to write some simple C++ programs.



Let's start by looking at a very simple C++ program. You saw a simple C++ program in the previous chapter and looked at what some of the statements might achieve, but the focus there was just on the layout and style. Now it's time to start understanding what the C++ statements actually do. Still, this program won't do anything especially useful just yet; the thing to concentrate on this time is just its general structure. Here it is:

```
// first.cc
//
// A Very Simple C++ Program
//
// Demonstrates the basic structure of a C++ program.
//
// Chapter 7.
//
// Author: AMJ
// Date: 17th October 2001
// Platform: g++, Linux

using namespace std;

int main ()
{

  int aNumber;

  aNumber = 0;

  return 0;

}
```

This is the *source code* of a program.

The first eleven lines in this program, starting with //, are *comments* (you saw some in the program in the previous chapter). The compiler ignores these lines; their only use is for a human reading the program. In this case they provide the name of the file containing the program, give a brief description of the purpose of the program, tell us who the author was and document when and on what system the program was written.

This is the *header block* of the program, a collection of comments that provides basic but important information about the program. A header block should, as a minimum, tell the reader who wrote the program, what the program does, why it was written, where it was written and when it was written. As a program is developed and maintained the header block should record the same information about all the changes. This will then provide all the information a human would need in order to understand how and why the program was originally written and how and why it has been changed since.

More generally, comments are used for many purposes; common uses are to explain parts of programs (especially complicated parts) and to record modifications to parts of a program. Their purpose is only ever for a human reading the program.

The line immediately after the header block:

```
using namespace std;
```

tells the compiler that this program is written using the standard C++ names and statements[1]. Your compiler may well let you miss this line out, but

1 This is not strictly true, but it's an explanation that will do just fine until you know a great deal more C++. The effect of this line is to tell the compiler that the program uses the `std` namespace, which will reduce typing by allowing you to use, for example, `cout` rather than `std::cout`. You really don't need to worry about this!

it's a good idea to include it to make your program more portable[2]. After all, you might at some point want to move your program to a computer where the compiler was stricter and required you to include it. This line, together with the comments, forms the preamble of the program.

The main body of the program starts at the next line:

```
int main ()
```

This marks the start of the program. Once the program has been compiled, this is the part of the program that the computer will execute first, the `main` function. All C++ programs must generate and return a single value to the computer when they exit, in the same way that all methods must generate and return a single value. The `int` in this opening line tells the compiler that this program will generate and return a single integer value (`int` is short for integer). This program does indeed do just that. The line:

```
return 0;
```

towards the end of the program produces this value. In this program the value returned is always 0.

It is a convention[3] that all C++ programs return a single integer value. This is almost always 0, a value that is taken to mean that the program has completed its task and exited successfully. The program is effectively telling the computer that it has finished and that all has worked as expected. Other values can be returned from more complex programs; these are normally used to send the message that some unexpected error has been encountered, so the program has failed to complete in the expected way.

You can treat this program as a template for your own programs. All of the lines explained so far will appear in every program you write. In fact, it's a good idea to keep them in a handy file that you can use over and over again, so that all you need do is change the comments at the top to explain a new program. It's also a fine idea to complete the parts of the header block in this file, such as your name, that will not change whatever the program you are writing.

The rest of the program is the *body*. The statements that make up the body of the program are contained between two *braces*, { and }. The programmer has indented the statements between the braces to emphasise that they are enclosed within them. This layout is ignored by the compiler, but is very helpful for the human reading the program. You have seen the importance of layout and style in the previous chapter.

There are two lines left. The first:

```
int aNumber;
```

is a *variable declaration*. This line tells the compiler that the program uses an integer value and that its name, more correctly called its *identifier*, is *`aNumber`*. Such values are called *variables* because the value they hold varies as the program is executed. Any program that does anything even remotely useful will use a lot of variables.

---

2 This line is required in standard C++. Most recent compilers should insist either that you include it or that you include `std::` before all the names in the `std` namespace. Including it is the easier option, especially if you don't understand what this means.

3 This convention is in fact now part of the ANSI standard for C++.

The second line in the program body is:

```
aNumber = 0;
```

This is an *assignment statement*. It simply assigns the value on the right-hand side of the = sign to the variable on the left-hand side. So here the variable *aNumber* is assigned the value 0. The = sign is not being used here in the sense that you are probably familiar with from mathematics or algebra; it is not a comparison, but an assignment. It's probably helpful to get into the habit of reading = as "becomes equal to" or "is assigned" rather than just "equals"[4].

These statements illustrate the two fundamental new ideas in this chapter. Variable declarations and assignment statements are two of the most fundamental ideas in programming. Programmers need to declare the variables that their programs will use and they need to be able to assign them values. The following sections cover these ideas in more detail.

Variables and types

A variable is a value used in a program. It is created in a declaration at the top of the program where it may also, if required, be given an initial value. The value then changes as the program progresses, as the statements in the program manipulate the value. A variable's final value is sometimes one of the outputs of the program. A variable has an *identifier* and a *type* of value that it can store. The variable's declaration tells the compiler what these two things are. For example:

```
int aNumber;
```

declares a variable with the identifier *aNumber* that can store integer values (in C++, `int`). It is good practice to provide an initial value for a variable at the same time as it is declared[5]. This assignment is simply added to the declaration. So, for example, to assign *aNumber* an initial value of 0 the declaration is extended to:

```
int aNumber = 0;
```

Declaring the type of a variable allows the compiler to allocate an appropriate amount of the computer's memory to store the variable's values, and allows the compiler to check that sensible operations are being carried out on values in the program. C++ has many data types, but for the moment just a few of these will be quite enough:

`int`	An integer value: 0, 1, 2, –1 and so on
`float`	A floating-point (decimal) number: 1.415, 1.25
`char`	A single character: 'a', 'Z', '$'
`string`	A sequence of characters: "hello", "Tony", "%$%!%%"
`bool`	A Boolean value: either true or false

There is an important distinction between the ways in which characters and strings are represented. Characters are enclosed by single quotation marks, but strings are enclosed with double quotation marks. This means that:

---

4 We will in fact see a different C++ symbol that is read as "equals" later on.

5 This is sometimes not necessary, but it is a good habit to get into. If in doubt you should always assign the variable an initial value.

```
'a'
```

is the single character *a*, with the type `char`, whereas:

```
"a"
```

is a string containing just one character with the type `string`. This distinction is subtle, but will become important.

A variable can only ever hold values of its own declared type. It is an error to assign, say, an integer value to a string variable. The reason for this should be quite obvious; the compiler has allocated the variable the right amount of memory to store an integer, and it's very unlikely that a string will fit in the same space.

Identifiers

The identifier chosen for a variable is pretty much irrelevant to the computer. The computer can't understand the language used for the name and certainly can't learn anything about the variable's purpose from it. However, the identifier is certainly very important for the programmer and for anyone else reading the program. It is essential that the name chosen should neatly describe the purpose of the variable. At the same time, the name should not be so long and descriptive that the program becomes unwieldy and hard to read. A programmer must strike a delicate balance between descriptiveness and brevity.

As an example, suppose a simple program was used to monitor the animals on a farm and that as part of this the program stored the number of sheep in a field. Here are some possible identifiers for that variable and some comments on them:

pigs	This is not a good name. In fact it's a very bad name because it's positively misleading!
x	At least this name isn't misleading, but it still doesn't tell the reader much about what the variable stores. This is poor.
thenumberofsheepinthefield	This name tells the reader what the value is but it's very hard to read. It's better than the last two suggestions, but a program using this would be an example of very bad style. It would also require a lot of typing!
sheep	This is better still, but it still doesn't really tell the reader what the value is.
numSheep	This is the name I would use and is the best of these suggestions. It's short enough to keep the program short and neat and it tells a reader what the value is. *numOfSheep* would be a reasonable alternative.

Choosing identifiers requires care and thought. The ones that you choose must provide some information for people reading your programs. Also, you might come back to a program after many months and have to understand it

quickly. Bearing this in mind, identifiers must also not be so long or unwieldy as to make the program unreadable. There is a balance to strike.

Most programmers have a convention that they use for their identifiers, or sometimes they will be asked to use one by their employer as part of some prescribed standards. Like programming style, the choice of identifiers is a personal thing and you will probably develop your own style for doing this as you gain more experience.

My style is to use a single word if possible or otherwise to combine two or three words and to capitalise the first letters of any words after the first (hence *numSheep* in the example). To save typing and to keep the program short and readable I abbreviate where the abbreviation is obvious; so number can become "num", value becomes "val" and so on. That's the convention I will use in the programs in the rest of this book. You're free to copy it, to follow your own convention, or to follow some other system that you might be required to use. Whatever you do, think about the identifiers you choose and use a consistent system for choosing them.

Declaring variables

Now it's time for a more detailed look at variable declarations. The basic form of a declaration is:

<type> <identifier>;

The declaration specifies the type of the variable (the type of value that it can store) and states its identifier (its name). The line finishes with a semi-colon. If the variable is to be given an initial value the declaration can be extended:

<type> <identifier> = <initial value>;

Here are some examples with comments to explain them:

```
int grade;      // an integer to hold a grade, no initial value
int grade = 0;  // an integer to hold a grade with initial value 0
char answer;    // an answer to something - one character
string name;    // a string to hold a name
string name = "Tony";   // as above, with an initial value
bool finished;  // to hold "true" if finished
bool finished = false;  // as above, but with an initial value
```

If more than one variable of a particular type is to be declared in a program, they can both be named on the same line:

```
int aNumber, anotherNumber;
```

This version is used especially when the two variables have closely connected purposes. It is identical in meaning to two separate declarations:

```
int aNumber;
int anotherNumber;
```

The choice between the two is largely a matter of the programmer's style and preference.

Values and variables

Variables aren't much use if a programmer can't give them values. Variables are assigned values in an *assignment statement*. The general form of such a statement is:

<identifier> = <value>;

The left-hand side of the = sign is the name of the variable that we want to assign a value to and the right-hand side is the value. The effect is that the value of the variable whose identifier is on the left becomes equal to the value that is on the right. For example:

```
// Declarations
int grade;
char answer;
string name;

// Assignments
grade = 70;
answer = 'a';
name = "alan turing";
```

These three statements assign the three values on the right to the variables named on the left. In each case the variable has to have the same type as the value assigned to it; it doesn't make sense to assign a string value to an integer variable. As examples of what cannot be done, assuming the types in the declarations above, these assignments are not allowed:

```
grade = "a*";       // cannot assign string to int
name = 100;         // cannot assign int to string
answer = "hello";   // cannot assign string to char
```

Unlike the algebraic use of =, it is not possible to write assignment statements with the variable's identifier on the right. This is not allowed and will cause a compilation error:

```
'a' = grade;        // Not allowed!
```

To make sure that you remember this, remember to get into the habit of reading the = sign in an assignment statement as "is assigned" or "becomes equal to".

In all the correct assignment statements so far the right-hand side of the assignment has been a simple value, called a *literal value*. It is probably more common for it to be an *expression*. An expression is something that calculates a new value. For example:

```
grade = 70 + 2;               // grade is 72
name = "ada " + "lovelace";   // name is "ada lovelace"
```

These expressions can also contain variables:

```
int sheep;
int pigs;
int animals;

sheep = 50;
pigs = 100;

animals = sheep + pigs; // animals is assigned 150
```

Finally, the variable that appears on the left-hand side can also appear on the right-hand side:

```
int sheep;

sheep = 50;

sheep = sheep + 1;  // sheep is now 51
```

You might remember that programming statement that I could never understand all those years ago! The one I mentioned in Chapter 3:

```
10 LET X = X + 1
```

This is, of course, an assignment statement (actually in the BASIC language, but it's very similar to the C++ equivalent). Its effect is to add one to the value of the variable *X* (sadly, most versions of BASIC only allowed single character variable identifiers).

The important thing to remember, especially when the same variable appears on both sides of the assignment statement, is that the value of the expression on the right is calculated first and the result is then assigned to the value of the variable named on the left. This means that a statement such as:

```
aNumber = aNumber + 1;
```

has the effect of adding one (called *incrementing*) to the value of *aNumber*. The right-hand side is calculated first, and the result is then assigned to *aNumber*.

Initialising a variable

When they are first declared variables have no defined value. They do have a value, but that value is undefined. This is what happens. The compiler allocates some of the computer's memory to store the variable. This memory has probably been used for something else before and will have some value left behind in it by another program. The initial value of the variable is whatever has been left behind in this memory, and this is something that it is quite impossible to predict with any confidence.

Using a variable without knowing its value can easily lead to incorrect results, so it is good programming practice to always give a variable an initial value immediately after it is declared:

```
int number;

number = 0;
```

This operation is so common that the two statements can be combined. This gives the extended form of a declaration that also includes an implicit assignment statement:

```
int number = 0;
```

This single statement has the identical effect to the two lines above.

Never assume that your variables have a value just after they have been declared. Get into the habit of initialising them. Never use a variable before it has been initialised, unless you are assigning an initial value to it, of course!

Operators and their precedence

Some of the examples of expressions above used the addition operation "+". Expressions in assignment statements can also contain all the other usual arithmetic operations (plus, minus, times, divide) that you might expect. The symbols might not be quite what you are used to in arithmetic. They are:

- / Division
- * Multiplication
- + Addition of numbers or concatenation of strings
- - Subtraction

When more than one of these is used in a single expression the operators have a *precedence* that determines the order in which they are executed. This is the same as you might be used to in arithmetic or algebra. The order of precedence for the four basic arithmetic operators is in fact exactly as in the table above – division is done first, then multiplication, then addition, and finally subtraction.

This often catches new programmers out and does indeed take a fair bit of getting used to. Look at the following statement:

```
int number = 2 + 8 / 2;
```

You might think that *number* is assigned the value 5. You might expect that 2 is added to 8 to give 10 which is divided by 2 to give the answer. Not so! The value *number* is actually assigned is 6. Division happens first (it has a higher precedence), so 8 is divided by 2 to give 4 and that result is added to the first 2 to give 6.

Since this can be confusing, brackets can be added to the expression to make things clearer. Again, this is the same as in algebra – the part of the expression in brackets is worked out first and then the precedence of the operators takes over. The statement above could be written:

```
int number = 2 + (8 / 2);   // number is assigned 6
```

to emphasise that the 8 is to be divided by 2 first. If this is not what is wanted, it could be written:

```
int number = (2 + 8) / 2;   // number is assigned 5
```

which changes the expression to make the addition happen first. In this case this also changes the result of the expression.

It is good practice to always include brackets in expressions even when the precedence of the operators would achieve the desired result. It makes the programmer's intention clearer and removes the possibility that other programmers will suspect a mistake!

Some shorthand

A common operation that happens in almost all programs is to add one to or *increment* the value of a variable holding a numeric value. For an integer variable the assignment statement for this is:

```
number = number + 1;
```

This is so common that C++ provides a shorthand form for this. A variable can be incremented using the ++ operator. If you remember that C++ is a

development of the programming language C you'll now know why it got its name. An integer variable can be incremented using this operator:

```
number ++;
```

This has exactly the same effect as the longer form:

```
number = number + 1;
```

The only advantage of using this shorthand is in the amount of typing the programmer has to do. The operation is so common that it does save a lot of typing.

Not surprisingly there is similar shorthand for taking one from (called *decrementing*) a variable's value:

```
number --;    // same as number = number - 1
```

Sometimes a variable must be incremented by more that one, so another shorthand is available that allows the programmer to add a value other than one to a variable:

```
number += 2;  // adds two to number
```

Equivalents exist to subtract a value, multiply by a value or divide by a value:

```
number -= 3;  // takes 3 off number
number *= 3;  // multiplies number by 3
number /= 3;  // divides number by 3
```

Remember that all these do is save typing. You can always write out the statement in full if you prefer, and many new programmers prefer to do so.

A warning about division

3 divided by 2 is sometimes 1. Really.

C++ provides several data types for storing numeric values. The two described so far are `int` (integers, whole numbers) and `float` (floating-point numbers). These two types of value are clearly both numbers and thus are the same sort of thing; it's often useful to be able to assign values of one type to the other. For example:

```
int number;
float cost;

cost = 12.0; // .0 indicates a floating-point value
number = cost;
```

could be used to assign the value of *`cost`* to the value of *`number`*[6]. This piece of program is fine. *`number`* now has the value 12. But suppose that *`cost`* had the value 12.75. The integer variable *`number`* can't store the decimal part, so the result would actually be 12; this is probably not what the programmer would have wanted. More often than not assignments like this are errors, so some compilers (mine included) will provide a warning when they encounter such statements.

This issue crops up most often in division. The thing to remember is:

---

6 Note that the `.0` on the value of `cost` isn't really necessary. It's included it here to clearly show that this is a floating-point value. Pedantically, it's a double-precision floating-point value.

an int divided by an int is an int and nothing but an int

This means that any decimal part of the result is lost. For example:

```
int number;
number = 5 / 2;   // number is 2
number = 1 / 2;   // number is 0
```

It might seem that this should not be allowed to happen and that any case where it does it an error. The fact is that sometimes this result can actually be what is required; it is actually a potentially useful result, the number of times the right-hand side of the division "goes into" the left-hand side. You might remember "remainders" in division; all that's happening here is that the remainder is being lost. Since this is likely to be correct, some compilers will not provide a warning of a possible error.

Statements could be written to calculate the result of a division including the remainder:

```
int number;
int remainder;

number = 5 / 2;                  // 2 goes into 5 twice
remainder = 5 - (number * 2);    // remainder is 1
```

Be careful when combining integer and floating-point values in your programs. If your programs are processing a lot of integer and floating-point values and are giving you incorrect results, always check for problems with integer division.

If you are dividing two integer values and you want the result to be a floating-point value you can force the result to be a float by using a *cast* to convert one of the integer values to a floating-point value, like this:

```
int aNumber = 12;
int anotherNumber = 5;
float result;

result = float (aNumber) / anotherNumber;
```

This would assign *result* the expected floating-point result, in this case 2.4. If you find yourself having to do this a lot it's probably time to consider whether the integer values shouldn't really be floating-point values.

An alternative, less elegant, way to convert an integer value to a floating-point value is to multiply the integer by 1.0:

```
result = (aNumber * 1.0) / anotherNumber;
```

This technique can be useful at times, but a cast is probably usually to be preferred.

Constants

Variables by their very definition vary. Sometimes a program needs something that behaves in the same way as a variable but that doesn't vary; this is unsurprisingly called a *constant*. Constants are declared in exactly the same way as variables except that the word `const` is added to the start of the line and the assignment of the value must be done at the same time as the declaration:

```
const int MaxTurns = 10;
```

The same guidelines apply for choosing identifiers for constants. Many programmers also choose to give them an identifier that clearly shows that they are not allowed to change. The convention that I use (and that is used in the programs in this book) is simply to use a capital letter to start the name.

Constants have two main uses. Sometimes they are used just for readability. Suppose a program made a lot of use of the mathematical constant *PI*. A programmer could write:

```
const float PI = 3.1415;
```

at the start of the program and then later use the constant in expressions such as:

```
area = PI * radius * radius;
circumference = 2 * PI * radius;
```

This makes the program much more readable. It also saves typing if the constant value is used in many places in the program and ensures that the value used is always precisely the same. It should be obvious that the statements above are far preferable to:

```
area = 3.1415 * radius * radius;
circumference = 2 * 3.1415 * radius;
```

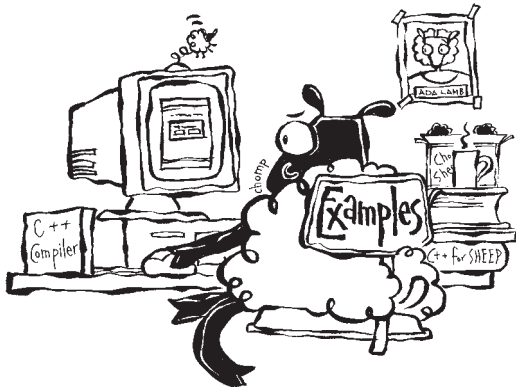
and that this a fine way to avoid errors or inaccuracies that might be caused by:

```
area = 3.14 * radius * radius;
circumference = 2 * 3.1415 * radius;
```

Another use of constants is when a particular value is used in many places in a program and might need to be changed later. The value will never be changed while the program is running, but it might change later for some other reason. Examples might be the number of games in a football season, the number of some item packed into a box, or the number of attempts allowed in a game. It's far easier to declare that value as a constant; if it changes you only have to change it in one place.

Variables and assignments

Even though there are only really two new ideas here, there is a lot to understand in this chapter. This is also the most important chapter in this book. Everything that follows has to assume that you understand how to use and declare variables and constants. Now, try the examples and exercises...



Example 1 – Farmer Strawstrap's farm

Farmer Strawstrap wants a computer program to help him keep details of the animals on his farm. He wants to keep details of the names and ages of all the animals. He also wants to know how many animals he has; on the farm at the moment there are many sheep and many pigs. What variables might his program use?

Taking things one at a time, the program will need variables for all the names of the animals and for all their ages. It will also need variables for the total number of sheep and the total number of pigs and, perhaps, the total number of animals. The names of the animals are strings and all the other values are integers.

Part of the variable declarations might look like this:

```
string sheepName;
string pigName;
int sheepAge;
int pigAge;
int numSheep;
int numPigs;
int numAnimals;
```

In the program itself there would be a line that calculated the total number of animals on the farm:

```
numAnimals = numSheep + numPigs;
```

Example 2 – Flossy's pocket money

Flossy wants to use a computer program to keep track of her pocket money. She wants to know how much she's got left at any time, so she decides that she needs to record how much she receives and how much she spends. What variables is the program going to need?

Money is a floating-point value[7], so all the values in Flossy's program would be declared as `float`. There would be a variable for the amount of money that she's currently got and another to store anything she spends. These would be

7 An issue here is that a `float` in C++ can store many places of decimals, but money values only ever have two decimal places. This can lead to errors, but only when a great many values and calculations are involved – we'll assume that Flossy doesn't get enough pocket money to make this a serious problem.

used in a suitable expression to update the amount she's got after she spends some.

The amount that she receives is unlikely to change very often, but it will probably change sometimes, so a constant would be used for that. The declarations might look like this:

```
const float WeeklyPocketMoney = 15.00;

float pocketMoney;
float amountSpent;
```

In the program there might be assignment statements like this:

```
pocketMoney += WeeklyPocketMoney;
```

This is of course equivalent to:

```
pocketMoney = pocketMoney + WeeklyPocketMoney;
```

Another assignment would alter the amount Flossy has after she has spent some:

```
pocketMoney = pocketMoney - amountSpent;
```

Remember that this last statement could also have been written:

```
pocketMoney -= amountSpent;
```

See how this shorthand saves typing!

Example 3 – Flossy's chocolate poser

Flossy and her friends all like chocolate. Because he likes to keep his animals happy Farmer Strawstrap gives them a box of chocolate bars to share every week. The number of bars in the box changes quite often so, in the interests of fairness, Flossy has written a computer program to work out how many bars[8] each sheep should get. Her variable declarations look like this:

```
int numSheep = 15;
int numBars = 50;
```

Flossy works out how many bars each sheep should get like this:

```
int barsPerSheep = numBars / numSheep;
```

When she runs the program it tells her that each sheep should have exactly three bars, so she shares them out. She is surprised to find some bars left over in the box. What's going on?

Flossy has mixed up her data types. She hasn't noticed that the number of bars should be a floating-point number. Because all the variables in her assignment are integers, integer division has taken place and the decimals have been lost. The expression to work out how many bars to give out should look like this:

```
float barsPerSheep = float (numBars) / numSheep;
```

This would tell Flossy how many bars to give to each sheep and also how she should divide up the bars that remain.

---

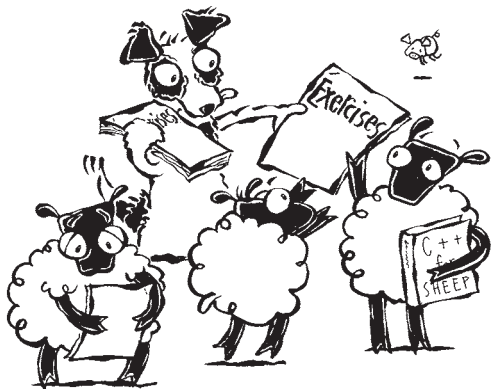
8 Baas? Sorry.

If it was not possible to divide the extra bars up, Flossy might use another assignment statement to find out how many bars are left over[9]:

```
int barsLeftOver = numBars - barsPerSheep * numSheep;
```

Operator precedence means that this statement would actually produce the required result, since the multiplication would be carried out before the subtraction. That said, it would be better to include brackets to show exactly what is intended:

```
int barsLeftOver = numBars - (barsPerSheep * numSheep);
```



7.1 Assuming that *number* is declared as an integer variable, what value is assigned to *number* by each of the following statements?

```
number = 10;
number = 10 + 6;
number = 10 + 6 * 4;
number = 10 + 6 * 4 / 2;
number = 10 + 6 * 4 / 2 - 2;
```

7.2 Assuming again that `number` is declared as an integer variable and that it has some value greater than 0 all of the following statements achieve the same thing. What?

```
number = number + number / number;
number += number / number;
number = number + 7 / 7;
```

7.3 Write two more statements that would achieve the same results as the statements in Exercise 7.2.

7.4 Identify and explain the error in the following fragment of a program.

```
int sheep;
char answer;

sheep = 10;
answer = sheep;
```

---

9 There is another arithmetic operator in C++, the modulus, represented by `%`. This effectively gives the remainder when one integer is divided by another. This line could also be written as: `float barsLeftOver = numBars % numSheep;`.

7.5 Identify and explain the error in this fragment.

```
string name;

name = 'Flossy';
```

7.6 A program stores the name of the favourite chocolate bar of a flock of sheep in a string variable. Which of the following is a good identifier for this variable?

thefavouritechocololatebarofthesheep
theSheepsFavouriteBar
colin
favBar
THESHEEPSTOPBAR

7.7 Assuming that *number* is declared as an integer variable, what effect do these statements have on the value of the variable?

```
number --;
// number = 10;
number *= 2;
```

7.8 Suggest the most likely data types to be used for a variable to store:

Your name
Your age
Your birthday
Your height
Your initials

7.9 Identify, explain and correct the errors in the following C++ program.

```
//
// A C++ Program

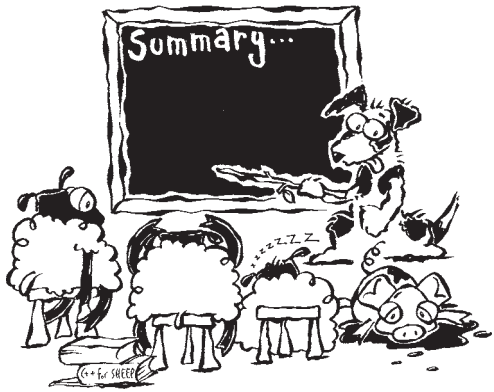
using namespace std;

int mane ()
{
  int number = 1.50;
  char letter;

  // float money;
  money = number;
  letter = '!';

  return 0;
```

7.10 Eggs are always sold packed in half dozens (packs of six). Write a C++ program that declares a variable to store the number of eggs available for packing and then calculates the number of boxes that are needed. You can assume that each box must contain exactly six eggs. Your program should also calculate how many eggs are left over. What would you have to change in your program if new regulations meant that eggs had to be sold in boxes of ten?



There has been a lot to learn in this chapter so you might want to go back soon and have another read! The ideas described here are fundamental to everything else in this book, so you should be sure that you're confident with them before reading on.

You should now know what a variable is and how one is declared. You should understand that a variable has a type and an identifier. You should know how to identify the appropriate types and choose good identifiers for the variables in your programs.

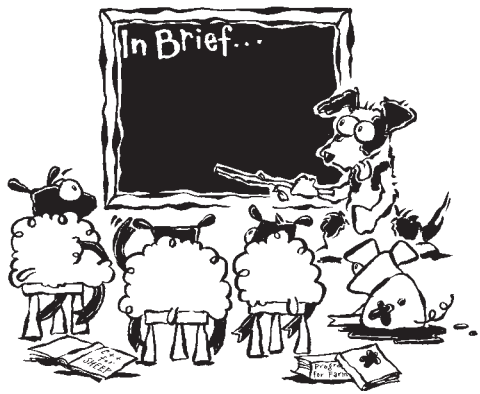
Constants are a special kind of variable that don't vary. You should be able to identify situations where it is sensible to use a constant in a program.

You should be able to write simple expressions using the basic arithmetic operators and you should understand the precedence of these operators.

The one thing that you can't do now is display the results of your program. Nor can you ask the users of your program to enter values for your program to process. Once we know how to do that we can start writing some really useful programs, so let's move on to the next chapter!



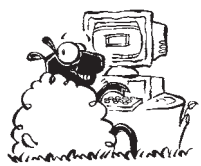

Chapter 8
Input & Output
Semaphore For SHEEP
cout
cin

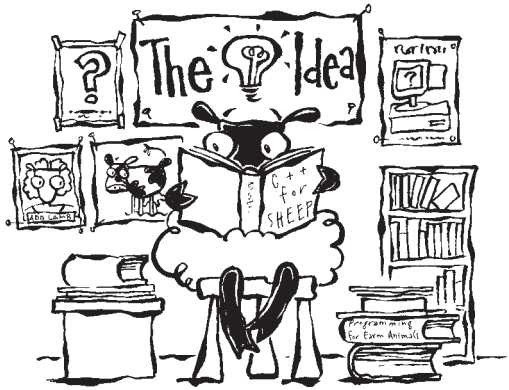


The programs that you have written and seen so far have not been especially useful. The first problem with them has been that all the values taken by your variables have had to be set in the program; it has not been possible to use values provided and entered by a user. This means that any changes to the values used by your programs would require changes to the actual source code, which would then have to be compiled. This is not something that the average user would want to do or would be able to do. Just as important, it has also not been possible to display the results produced by your programs in any place or form that the user might see them.

This chapter explains how to ask the user of your program to enter some values, how to read these values into your variables, and lastly how to display the final results of your program to the user. It also describes some very basic aspects of human–computer interaction; you will want your users to enter the values you require and expect, so you will need to ensure that they properly understand what they are being asked to do.

After reading this chapter you should understand how to output the results produced by your programs and how to request and read values input by the users of your programs. You should be able to design simple, yet effective, dialogues to allow your programs to interact with your users. You should now be able to write simple C++ programs that manipulate values to produce useful results.





In most books about programming or about a programming language the first program the reader sees is always the same. This first program always does nothing more than display the message *hello, world* or something very similar on the user's screen.

This is such a fine and longstanding tradition that it would be a shame to not to include this memorable program somewhere in this book, even if it isn't the first complete program. So here is my version of that very program in C++:

```
// hello.cc
//
// The Traditional First C++ Program
//
// Display "hello, world" on the screen.
//
// Chapter 8.
//
// Author: AMJ
// Date: 20th October 2001
// Platform: g++, Linux

#include <iostream>

using namespace std;

int main ()
{

  cout << "hello, world" << endl;

  return 0;

}
```

Although it's still very short this program contains several important new features.

The first new line is the first one after the initial comments that form the header block:

```
#include <iostream>
```

This line tells the compiler where to find the definitions of the C++ statements for input and output. Specifically, it instructs the compiler to include the `iostream` library where they are stored, so that they can be accessed. The `iostream` library contains the definitions of `cout`, `endl` and the other new C++ that you can see in the program and that we'll meet later in this chapter.

You'll remember from Chapter 2 that the linking part of the compilation process links the C++ system libraries with the program being compiled. It's hard to imagine many useful C++ programs that don't do any input or output, so you might well want to add this include line to your program template file.

The other line that contains C++ you haven't met before is:

```
cout << "hello, world" << endl;
```

As you've probably guessed, this line outputs the string *`hello, world`* on the screen:

```
hello, world
```

The string to be output is included between double quotation marks; the quotation marks themselves are not displayed. The `endl` at the end of this line adds a "return" character to the end of the line. This is similar to pressing the Return key when you're using a word processor or your text editor, and moves the cursor (in this case, the position where the next output would appear) down one line.

Books about programming are very predictable things. The task carried out by the first example program to request and use input from a user is almost as traditional as the task of the first program of all. This program prompts the user to enter their name and responds with a cheerful greeting message. Again, it would be a shame to deviate from such a fine tradition, so in C++ this program looks like this:

```
// greeting.cc
//
// The Traditional Second C++ Program
//
// Display a cheerful personal greeting on the screen.
//
// Chapter 8.
//
// Author: AMJ
// Date: 20th October 2001
// Platform: g++, Linux

#include <iostream>
#include <string>

using namespace std;

int main ()

{
  string name;

  cout << "Please enter your name: ";
  cin >> name;
```

```
    cout << "hello, " << name << endl;

    return 0;

}
```

The first new line in this program is another `#include` line. This time it's telling the compiler and linker to include the `string` library. This provides everything that allows the program to use and manipulate `string` variables. Here, a `string` variable (with the identifier *name*) is declared in the first line of the `main` function.

The first `cout` line displays a *prompt* to the user inviting them to enter their name. The `cin` line immediately below reads what they type at the keyboard up until they press the return key. The >> and << symbols indicate that values are flowing into or out of the program. Finally, the second `cout` line in the program displays the required cheerful greeting.

When being run, the program dialogue would look something like this:

```
Please enter your name: Tony
hello, Tony
```

That's a quick look at how input and output work in C++. The next sections describe what's happening here in some more detail.

Input and output streams

In C++ input and output are achieved via `cin` and `cout` respectively. By default, `cin` reads the data that the user types at the keyboard and `cout` displays output on the user's display (usually the computer screen). The proper C++ name for `cin` and `cout` is *streams*; `cin` is the input stream and `cout` is the output stream[1]. You can think of the information to be input or output as "flowing" through these streams; the accompanying >> and << operators point in the direction the values are flowing, either into or out of the program.

To use `cin` and `cout` a program must include the `iostream` (Input and Output Stream) library. This is done with the line:

```
#include <iostream>
```

which must appear before the start of the main part of the program. It is usual to put it together with any other `#include` statements immediately after the heading comment block. This statement instructs the C++ compiler and linker to include the built-in libraries for input and output.

`cin` and `cout` work in similar ways. Each has an operator (>> for `cin` and << for `cout`) that indicates the values that they should process. Each operator looks like an arrow and shows the way in which the values are "flowing". Take care not to mix these operators up! `cin` is always used with >> and `cout` is always used with <<. You could in fact remember that the C++ command for input is `cin >>` and that for output is `cout <<`.

Input using `cin`

`cin` (or, if you prefer, `cin >>`) reads data from the keyboard. It reads data when the user has pressed the return key and not before. The value entered is then

---

1 Another stream, `cerr`, is also available. It is the standard error stream, and is used specifically for error messages. It works in the same way as `cout`.

passed into the variable provided after the >> operator. A simple example reads an integer:

```
// Declare and Read an Integer

int number;

cin >> number;
```

Other data types work in the same way, except that it is not possible to read a Boolean value from the keyboard. If a string is being read it is read up until the first *white space* (a space, return or tab) but only after the Return key is pressed. Any characters after the first white space are ignored and left for the next `cin` statement to read. For example, suppose a program includes the statements:

```
// Declare and Read a String

string name;

cin >> name;
```

This would read a string into the variable *name*, but only the part of the string before the first white space. For example:

String input	Value of *name*
alan	*alan*
alan m turing	*alan*
alan?	*alan?*
alan ?	*alan*

The remaining unread input could be processed if the program included another `cin` statement:

```
// Declare and Read a String

string name, name2;

cin >> name;
cin >> name2;
```

String input	Value of name	Value of name2
alan	*alan*	Undefined
alan m turing	*alan*	*m*
alan?	*alan?*	Undefined
alan ?	*alan*	*?*

In these examples, where only one string is provided the program would wait until some more input values were entered (it requires at least one more string before it can continue). It is quite likely that the user would have no idea what was going on in this case, so there is a clear need for some mechanism to keep the user informed of what is going on.

The problem is that in all these examples of input the user has no idea of what they are being expected to enter! Effective programs should make use of

well thought-out dialogues with users and as a minimum should provide a meaningful prompt whenever a value is to be entered. Prompts are displayed using `cout`.

Output using `cout`

`cout` (or `cout <<`) works in much the same way as `cin`. Each `cout` statement specifies the data to be output using the `<<` operator. `cout` can output either the value of a variable or a string of characters.

A string of characters to be output is enclosed with double quotes:

```
cout << "hello, world";
```

The value of a variable is output by providing just the variable's identifier. For example, to output the value held in the variable *`number`* the `cout` statement is:

```
cout << number;
```

This is not, of course:

```
cout << "number";
```

which would output the string *`number`* rather than the value of the variable with that identifier:

```
number
```

The double quotes that surround the string are never output. If you do want to output a double quote in a string you can put a *backslash* character before the quote:

```
cout << "\"hello, world\"";
```

This looks rather cumbersome, but it would output:

```
"hello, world"
```

Some systems may also require that a backslash character be used before a single quotation mark, as in:

```
cout << "hello, it\'s a fine day";
```

which would display:

```
hello, it's a fine day
```

The *Local guide* or a simple experiment will show you whether your system requires this. Alternatively, you can take the view (as I tend to) that the backslash does no harm and you can just include it anyway.

A single character can be surrounded by single quotes, since it has the type `char` rather than the type `string`:

```
cout << "The first letter of the alphabet is " << 'a';
```

which outputs:

```
The first letter of the alphabet is a
```

There could, of course also be:

```
cout << "The first letter of the alphabet is "
     << "\'a\'";
```

in which case there is more than one character, making the second part a string!

```
The first letter of the alphabet is 'a'
```

This is not something to worry about, especially since it works just as well with double quotes. It is basically a matter of style. I generally use a single quote for character variables, but it's up to you.

This last example also illustrates in passing another point of style. Where a `cout` statement spreads over more than one line, it is usual to align the `<<` at the start of each line to make sure that the layout is suitably neat.

Output elements can be combined with the `<<` operator, as in this example. So to output the value of a variable together with a descriptive message the statement would be:

```
cout << "The value of number is " << number;
```

Finally, it is sometimes necessary to add a new line character to the end of an output statement. This has the same effect as the user pressing the Return key and just moves the output on to the next line. For example:

```
cout << "This is on the first line" << endl
     << "and this is on the second" << endl;
```

which would display:

```
This is on the first line
and this is on the second
```

This can be compared with:

```
cout << "This is on the first line"
     << " and so is this" << endl;
```

which would display:

```
This is on the first line and so is this
```

`endl` (that's `end` followed by a letter `l` and not a digit 1) is only ever used with `cout`. It is never used with `cin` (where it is meaningless anyway) and trying to use it with `cin` will generate a compilation error. Nor is it used with every `cout` statement; it should be used only when you want the output to move to the next line.

A particular case when `endl` should probably not be included is when asking the user to enter a value at a prompt. If the user has to enter the value below the prompt the dialogue can look very ugly and the input is moved away from the prompt itself. For example, this code is an example of poor dialogue design:

```
cout << "Enter your name" << endl;
cin >> name;
```

When run, the dialogue would be:

```
Enter your name
Tony
```

which is rather ugly. Far neater in this case is not to use the `endl` at all, and perhaps to add something such as a colon to the end of the prompt to show the user where to type:

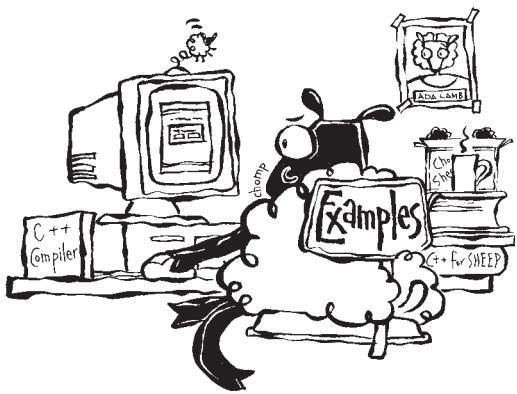
```
cout << "Enter your name: ";
cin >> name;
```

which looks rather better when run:

```
Enter your name: Tony
```

Input and output

Now that you can use input and output you should be able to write some much more useful C++ programs. So, on with the examples and exercises!



Example 1 – Farmer Strawstrap's farm revisited

Farmer Strawstrap still wants a computer program to help him keep details of the animals on his farm. He particularly wants to know how many animals he has. He knows that on the farm at the moment there are many sheep and many pigs, but they keep moving around and he keeps losing count. He also wants to know what percentage of his animals are sheep and what percentage are pigs.

Write a program that prompts Farmer Strawstrap to enter the number of sheep and the number of pigs and then displays the total number of animals on his farm together with the percentage that are sheep and the percentage that are pigs.

As you'll probably remember from the exercises in the previous chapter, this program will need two integer variables. One will be used to store the number of sheep and one will store the number of pigs. A third variable will be used to store the total number of animals. The percentage figures are floating-point values, so we will have to be careful that integer division does not cause incorrect values to be produced. A cast will be used to force a floating-point result.

The program will follow a very basic pattern. It will prompt the user for the required input values, it will then do some calculations, and it will finally display the results. This is a pattern that you will meet again and again and is, in fact, the basic pattern followed by almost all programs.

These three phases are shown by comments in the code. The complete program looks like this:

```
// animals.cc
//
// Calculates the number of animals of the farm and
//   displays the percentages that are sheep and pigs.
//
```

```
// Chapter 8, Examples 1.
//
// Author: AMJ
// Date: 20th October 2001
// Platform: g++, Linux

#include <iostream>

using namespace std;

int main ()
{
  int numSheep, numPigs, numAnimals;
  float percentSheep, percentPigs;

  // Get Input Values

  cout << "Enter the number of sheep: ";
  cin >> numSheep;

  cout << "Enter the number of pigs: ";
  cin >> numPigs;

  // Calculations

  numAnimals = numSheep + numPigs;

  percentSheep = (float (numSheep) / numAnimals) * 100;
  percentPigs = (float (numPigs) / numAnimals) * 100;

  cout << "There are " << numAnimals << " animals"
       << endl
       << percentSheep << "% are Sheep" << endl
       << percentPigs << "% are Pigs" << endl;

  return 0;

}
```

Example 2 – Flossy's pocket money revisited

Flossy still wants to use a computer program to keep track of her pocket money. She wants a program that will add the amount she receives to the amount she has left and then subtracts the amount she has spent.

This program will follow the same pattern as Farmer Strawstrap's. There will be input, then calculation, and finally output. It will prompt Flossy to enter the amount of money she has left over and the amount she's spent. It will then display how much she has left. It will also add in the amount that she has received for the week.

The program is quite straightforward:

```
// money.cc
//
// Flossy's Pocket Money Program
// Calculates the amount remaining after expenditure.
//
// Chapter 8, Examples 2.
//
```

```
// Author: AMJ
// Date: 20th October 2001
// Platform: g++, Linux

#include <iostream>

using namespace std;

const float WeeklyPocketMoney = 15.00;

int main ()
{

  float pocketMoney;
  float amountSpent;

  // Input

  cout << "How much is left from last week? ";
  cin >> pocketMoney;

  cout << "How much have you spent? ";
  cin >> amountSpent;

  // Calculations

  pocketMoney += WeeklyPocketMoney;
  pocketMoney -= amountSpent;

  // Output

  cout << "You now have " << pocketMoney
       << " left." << endl;

  return 0;

}
```

Example 3 – Flossy's chocolate poser revisited

You will remember that Flossy and her friends all like chocolate and you will also remember that Flossy uses a C++ program to work out how many bars of chocolate each of the sheep should get. Her program prompts her to enter the number of sheep and the number of bars available, and then displays the number of bars that she should give to each sheep. It also displays the number of bars that she will have left over.

This program again follows the pattern of the other two. Integer variables will be used to store the number of chocolate bars and the number of sheep. In this case integer division is actually required, since the bars cannot be split. A simple calculation after the division will determine the number of bars left over. The program first works out how many bars each sheep should get and then multiplies this value by the number of sheep to find how many bars are left over.

```
// chocolate.cc
//
// Shares chocolate bars between the sheep and displays
//   how many each sheep should get and how many bars
```

```
//   are left.
//
// Chapter 8, Examples 3.
//
// Author: AMJ
// Date: 20th October 2001
// Platform: g++, Linux

#include <iostream>

using namespace std;

int main ()
{

  int numSheep;
  int numBars;
  int barsPerSheep;
  int barsLeftOver;

  // Input

  cout << "Enter the number of Sheep: ";
  cin >> numSheep;

  cout << "Enter the number of Chocolate Bars: ";
  cin >> numBars;

  // Calculations

  barsPerSheep = numBars / numSheep;
  barsLeftOver = numBars - (barsPerSheep * numSheep);

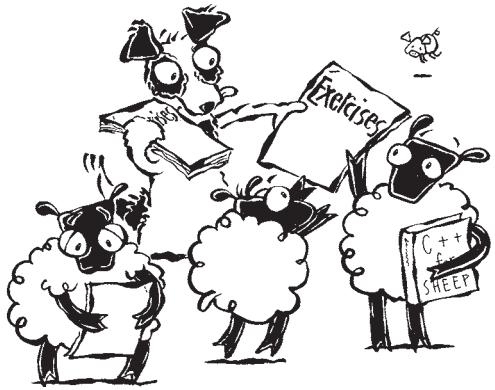
  // Output

  cout << "Each sheep should have " << barsPerSheep
       << " bars. There will be " << barsLeftOver
       << " bars left over." << endl;

  return 0;

}
```





8.1 Identify and explain the errors in the following C++ program. When you have found the errors, create a file containing a correct version of the program and compile and run it on your system.

```
#include <iostram>

using namespace std;

int main ()
{
  string name;

  cout << Enter Your Name:
  cin >> name >> endl;

  cout << "hello, name" << endl;

  return 0;

}
```

8.2 What is the output of the following fragment of program?

```
int number = 10;

cout << "The value of number is " << number << endl;
```

8.3 And what is the output of this? Be precise!

```
string name = "John von Neumann";

cout << "\"" << name << "\" is very famous!" << endl;
```

8.4 Write a program that prompts the user to enter the heights of five sheep and then calculates and displays the average height of the sheep. Assume that the heights are integers and are provided in whole centimetres. What data type is the average?

8.5 Write a program that prompts the user to enter the heights and names of two sheep. It should display the average height and the difference in height between the sheep. Assume that the details of the tallest sheep are entered first. The final output should look something like:

Flossy is 5 cm taller than Freda.

```
Their average height is 63.5 cm.
```

8.6 Write a program that prompts the user to enter their name, the year in which they were born and the current year. The program should then display a message similar to:

```
Hello, John. It is 2003 and you will be 27 this year!
```

8.7 Today in 2045 I will receive a free bus pass. Write a program that will prompt me to enter the current year and will tell me how many days, hours, minutes and seconds there are until today in 2045. Assume that there are 365.25 days in each year.

8.8 Write a program that reads a user's full name and displays a cheerful greeting. The program should read the forename and surname separately (assume that each is only one word) after suitable prompts. The final greeting should be something like:

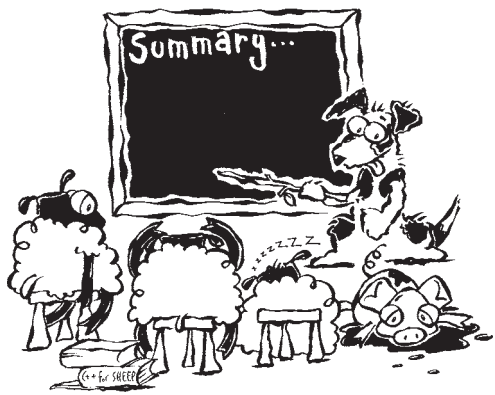
```
hello, Alan Turing!
```

8.9 Modify your program from Exercise 8.8 so that the output is:

```
I would just like to say "hello, Alan Turing!"
```

8.10 Modify your program from Exercise 8.8 once more so that the output is:

```
Turing is very famous. His/her first name was Alan.
```

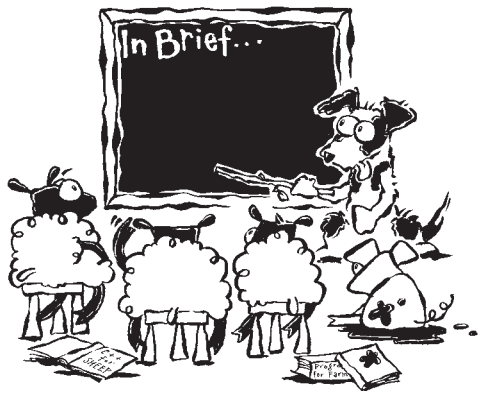


This has been the second of two very important chapters. Together with the previous chapter, it has introduced the very basics of C++. You should now be able to write programs that take a user's data, process the data, and produce and display useful results. You can write C++ programs that perform everyday calculations – some of these programs could actually be useful!

You should now understand how to use `cin` and `cout` to perform input and output. You should be able to use `cin` to read values into variables, and `cout` to display the values of the variables. You should also understand that you must include the `iostream` library if you are to use `cin` and `cout`.

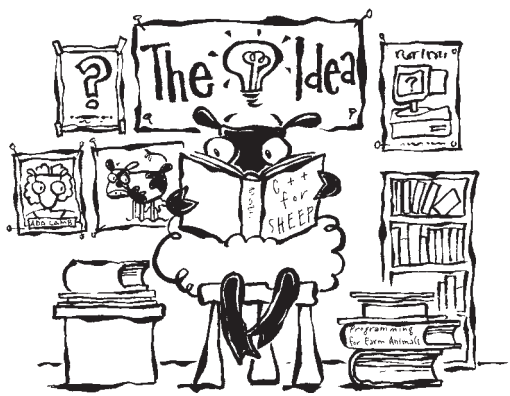
Now we will start to look at some more C++. Everything that we will do from now on will rely on variables and input and output. Soon we will be writing some really useful programs!


Chapter 9
A word on Testing
Done
Test Plan



You have now written your first complete C++ program. Most of the rest of this book will be about C++, but in many ways writing a program is only half of the story. If a program is going to be truly useful, the programmer and, more importantly, the program's intended final users must have confidence in the results that it produces. They must be sure that the results they see from the program are correct. Business users are potentially going to base costly business decisions on the results produced by the programs they use, pilots are going to trust navigational systems in their aircraft, and so on. This brings us to a brief pause from looking at C++. This chapter looks briefly at how and why computer programs are tested.

After reading this chapter you should understand something about the ways in which all your programs should be tested. You should be able to write a *test plan* for a program and you should also be able to carry out the plan. Using the ideas from this chapter you should be able to demonstrate that all the C++ programs you write in the future are "correct" and you should understand why it is never possible to be entirely sure that a program is completely correct!



Writing a computer program is a complicated business, but it is far from the only activity involved in producing reliable software for reliable computer systems. We have already seen the importance of thorough analysis and design before a program is written. The process of producing reliable computer systems is usually called *software engineering*. This term includes a wide range of activities – designing software, testing software, writing

documentation and more – of which programming is only quite a small part. Some programmers call themselves *Software Engineers* to reflect the range of tasks involved; in many organisations a programmer would be expected to carry out a range of such tasks. Programming and software engineering are not the same thing.

Computer programs are everywhere in the modern world. Modern cars have systems that rely on extremely complex programs to control everything from the car's heating system to its brakes. Airliners have computerised "fly-by-wire" systems that almost allow them to navigate and fly themselves. Anyone who drives a car or flies on an airliner is placing a lot of trust in these computer programs and by implication in the skills of the programmers who wrote them; if the programs failed by producing the wrong results the consequences could be catastrophic. Remember that all these programs were at some point written by human programmers and that humans often make mistakes. That is why all these programs were also very thoroughly tested before anyone was allowed to use them in any sort of potentially costly or dangerous situation.

Testing programs is another important part of software engineering. For a computer system to be truly useful its users must be confident that the system functions correctly and that the results it produces are correct and reliable. The process of testing a computer program to ensure its accuracy and reliability is in many ways much more important than the process of writing the program itself.

Testing a simple program

Flossy's Chocolate Sharing program from the previous chapter will serve as an example of how a simple program can be tested. This program shares a number of chocolate bars between a number of sheep and displays how many each sheep should have and how many bars are left over[1]. The process of testing this program thoroughly is quite complex, even though the program itself is very small and straightforward.

Here is the program again:

```
// chocolate.cc
//
// Shares chocolate bars between the sheep and displays
//   how many each sheep should get and how many bars
//   are left.
//
// Chapter 8, Examples 3.
//
// Author: AMJ
// Date: 20th October 2001
// Platform: g++, Linux

#include <iostream>

using namespace std;

int main ()
```

1 Admittedly, the results of an error in this program are unlikely to be as catastrophic as, say, an error in a fly-by-wire system, but an enraged sheep that has lost out when the chocolate is being shared out is not a pleasant thing.

```
{

  int numSheep;
  int numBars;

  int barsPerSheep;
  int barsLeftOver;

  // Input

  cout << "Enter the number of Sheep: ";
  cin >> numSheep;

  cout << "Enter the number of Chocolate Bars: ";
  cin >> numBars;

  // Calculations

  barsPerSheep = numBars / numSheep;
  barsLeftOver = numBars - (barsPerSheep * numSheep);

  // Output

  cout << "Each sheep should have " << barsPerSheep
       << " bars. There will be " << barsLeftOver
       << " bars left over." << endl;

  return 0;

}
```

The first step in testing this program is to develop a plan; not surprisingly this is called a *test plan*. This plan will simply list some input values (called *test data*) that will be provided to the program together with the results that would be expected if the program were working as expected. These expected results are calculated, in advance, by hand. Each set of input values and expected results forms one *test case* for the program. If the program produces the correct results for each test case (and assuming that there are sufficient test cases to cover all possibilities) then it will be safe to assume that the program is correct and so works as intended.

Some sample test data for Flossy's program is:

Inputs		**Expected outputs**	
Sheep	**Bars**	**Bars per sheep**	**Bars left over**
10	100	10	0
10	105	10	5
12	100	8	4

So, for example, the third test case here states that if there were 12 sheep and 100 bars the correct results from the program would be eight bars for each sheep with four bars left over. The other two test cases simply test other combinations of input values.

Testing the program is then just a case of running it for each set of data and recording the actual results. These results are compared to the expected

results for the same test case and if they all match the program has passed all the tests and is assumed to work.

The problem is that this doesn't mean that the program actually works! There is only sufficient information here for a programmer to assume that it does. In the example above there are only three sets of test data; if the program passes all three test cases, all that is actually known is that it works correctly with these three pairs of input values. It is certainly not possible to be sure that it works with every possible set of input data; with most types of input data[2] there could always be some extra test case that has not been tried for which the program produces an incorrect result.

This is the problem with testing any program. A program will be expected to work with very many input values; a program very similar to the program above could potentially process an infinite number of possible values. For this reason it is rarely possible to test any program with every possible set of test data. Instead, representative values are used to build up test cases in a test plan; if the program passes all the test cases it is reasonable to assume that it works correctly in all other possible cases. This is the case with all programs; a program in an airliner cannot be tested on every route the airliner might fly and in all weather conditions. For any program a subset of the possible input values must be used in the test plan. Picking those values requires thought; this process cannot be allowed to be random.

Programs should also be able to cope with input values that are not as expected; a user may enter a negative number when a positive number is expected[3], for example. This means that programs should also be tested with unexpected values. In this example an input that indicated that there was a negative number of sheep or bars is obviously nonsense, but it is important that the program deals with such inputs sensibly. There are also some specific cases to look out for. There is a potential error in this program if 0 were entered for the number of sheep. The number of bars would then be divided by 0, and dividing any positive number by 0 gives infinity, which is in effect an error. The program must also cope with this error and must behave sensibly.

If we take these potential error cases into account, a more complete test plan for the example would look something like:

Inputs		Expected outputs		Pass?
Sheep	**Bars**	**Bars per sheep**	**Bars left over**	
10	100	10	0	
10	105	10	5	
12	100	8	4	
–1	100	Error		
10	–1	Error		
0	100	0	100	

2 In this small example it might be possible to test all the possible values, assuming that there is a small number of sheep and a manageable number of chocolate bars. But, in more sophisticated programs, there is generally no sensible way to record and test all the possible input values.

3 Users are like that.

Once again there are assumptions in this plan. A new one is that if the program correctly handles one input of a negative number for each input value it will also correctly handle all others. It is also assumed that the program correctly handles an input of 0 for the number of sheep with any value for the number of bars. These assumptions (and there are more even in this simple case) are necessary, since it is rarely feasible to test a program with all its possible combinations of input values. A final change in this test plan is the addition of the final column; this is just there to record the result of the tests.

Boundary and typical values

A systematic plan is needed for choosing the values to be used in test cases. One possibility would be to just choose random values[4], but this could lead to carrying out too many or too few tests and could mean that the testing takes far too long or overlooks some important test cases. Something much more systematic is required. A system is needed that will allow testers to choose a small enough number of test cases so that they can actually carry out the test in a reasonable amount of time, but which still gives them enough test cases to allow them to be confident that a program is correct.

There are basically two sorts of expected values that can be input to a program. *Typical values* are simply examples of those that the program is expected to process when it is running normally; testing with these is just a case of choosing a collection of typical values and assuming that if the program processes these all correctly it will also correctly process all other similar values.

Boundary values are more complicated. These are those that occur around a limit on the acceptable values. In the example above there is a lower boundary on the possible value for the number of sheep – it cannot be less than one. The program should be tested with all boundary values and also the two values immediately surrounding each boundary. In the example there is an assumption that there is no upper boundary on the number of sheep, although there might well be in practice.

To help the tester, a test plan should also indicate the purpose of a particular test case – whether the value is typical or boundary. A full test plan for Flossy's Chocolate Sharing program might be:

	Purpose	Inputs		Expected outputs		Pass?
		Sheep	Bars	Bars per sheep	Bars left over	
1	Typical	10	100	10	0	
2	Typical	10	105	10	5	
3	Typical	12	100	8	4	
4	Boundary	2	100	50	0	
5	Boundary	1	100	100	0	
6	Boundary	0	100	Error		
7	Boundary	10	1	0	1	
8	Boundary	10	0	0	0	
9	Boundary	10	–1	Error		

4 Programs do exist that generate random numbers that can be used as test data to be used as input to other programs. They are useful for carrying out large numbers of tests automatically.

The other addition this time is numbers to index each test case. This is simply for ease of reference, so that, for example, the testers could easily and unambiguously give the programmers a list of the tests that the program fails.

In this plan, tests 1 to 3 are randomly chosen typical values, tests 4 to 6 test the lower boundary on the number of sheep (we have assumed that there must be at least one sheep and that there is no upper limit), and finally tests 7 to 9 test the boundary on the number of bars (this time there can be no bars at all, but there cannot be a negative number). Notice how most of the testing is concentrated around the boundary values.

Writing and using the test plan

The test plan can be written directly from the specification of the program, so it can and should be written before the program. It is normally easy to identify boundary and typical values from the specification, and the expected outputs should be calculated by hand and carefully verified. The plan should be allowed to be dynamic to some extent; it should be changed if the program's specification is changed. The plan should certainly be complete before starting to write the program. While some testing will take place during the program's development as the programmer checks the code, the final test should always be carried out according to the plan. The final test of a program should never be carried out randomly and the final test plan should never be written after the program is complete.

In many commercial software development businesses there will be people whose specific job it is to test programs. These will rarely be the same people who write the programs. This is a good thing; it is sometimes very difficult to be objective when you test your own programs. After all, you wrote it and you know how good it is. You know it works, so what is the point in thoroughly testing it?

It's likely, though, that while you learn to program you will be testing your own programs. When you do this, try to be as thorough and as ruthless as possible. It's worth remembering that if you don't find an error someone else will. Make sure that your test plan is complete and that it covers all the possible cases. It's also a good idea to ask a friend to test your program while you return the favour and test theirs. You can enjoy yourself finding tiny errors in your friend's program while they derive similar satisfaction from yours.

The final version of a program should pass all the tests in the plan. Sometimes, of course, the plan will show up errors that must be corrected and the program will have to be changed as a result. Obviously the program must then be tested again. In this case it is vital that the whole test plan is run again after any change; a change to correct one error may well have caused an error somewhere else in the program, and another test case may now produce incorrect results. The final version of a program should always pass every single test case in the plan.

Testing and debugging

As you write your programs you will be running them from time to time to check that they compile and work as expected. More often that not you will look at the outputs, spot mistakes and then make changes to the program. You will find bugs and you will fix them. This is *debugging*, not testing.

Do not confuse these two activities. Debugging is an important part of the programming process; a programmer finds errors in the program and fixes them. But this happens before testing. Testing is a structured activity used to verify that a program is correct. It takes place only when the program is believed to be correct and when all the known bugs have been found and corrected. Of course, testing may well uncover more bugs, but it remains a separate activity.

User testing

The first true test of any computer program comes not as it is being written but when it is used "in anger" by one of the intended users. This kind of testing takes place in a very different environment from that in which the testing strategy described here is used. Users are very unpredictable things[5] indeed!

Most commercial software will be thoroughly tested by the software company that wrote it. But before it is made widely available the company will always send it to a few selected users for *beta testing*. This testing simulates the environment in which the programs will eventually be used and can often show up errors that could not be found with any test plan.

You can adopt this idea, again by cooperating with a friend. A fine way to test one of your programs is to ask a friend[6] to use it for a while. Don't give them the test plan and don't ask them to test it systematically, but just ask them to play around with your program and see what they can find. The idea is to see how your program works when a real user uses it. Challenge your friend to find a mistake in your program; you might even offer them a prize if they find one! You will be amazed at what they will do with your program as they attempt to break it.

Testing is important

Testing a program thoroughly can be a tedious and repetitive job, but it is also a vitally important one. A program that produces incorrect results is a very dangerous thing; many people have great faith in the figures generated by a computer and errors in these figures can be costly and dangerous. Whenever we get in a car or fly in an airliner we are placing a great deal of faith in the programmers who wrote the programs, and perhaps even more faith in the testers who tested the programs.

Whole books have been written about testing, and this chapter has been only a very quick taster of the whole process. You should now be able to write a test plan for each program you write and you should be able to carry it out. Above all, you should understand why thorough testing is so important!

You should also realise that it is never possible to be completely certain that any program works as expected in all cases. There will always be some set of input values, some test case, or some other set of circumstances that might cause the program to fail. What is important is that the testing is as thorough as possible, so that the users can have as much confidence as possible in the

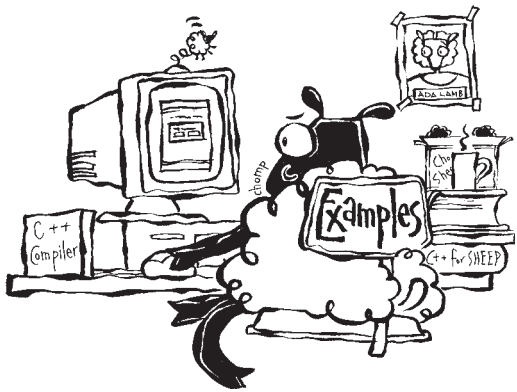
---

5 It has been said that the most used expletive in the software development industry is "Users!". It is probably true.

6 A good friend. Or at least a friend you trust. Friendships can be broken over errors in programs!

program. Obviously the program's application will also determine how thorough the testing must be; a program to share chocolate bars probably requires less thorough testing than a program to fly an airliner.

In fact, testing a program thoroughly is just as important as writing it in the first place. In many cases it is much more important; an untested program is a useless and sometimes even dangerous thing.



Example 1 – Ice skating sheep

Flossy and her friends hold an annual ice skating competition. Each competitor receives a mark out of six from each judge. There are five judges. The final mark for the competitor is calculated by discarding the highest and lowest mark received and then taking the average of the three marks that remain.

Write out a test plan for the program that calculates the final marks.

The highest mark from any judge is 6 and the lowest is 0, so these are the boundary values. The plan will also have to verify that the correct average is calculated and that the correct scores are being disregarded; this can be done with a range of typical values.

The plan would be:

		Judge						
	Purpose	**1**	**2**	**3**	**4**	**5**	**Mark**	**Pass**
1	Typical	6	6	6	6	6	6	
2	Typical	1	6	3	3	3	3	
3	Typical	6	6	2	2	4	4	
4	Typical	1	6	2	4	4	3.33	
5	Boundary	–1	6	6	6	6	Error	
6	Boundary	0	6	6	6	6	6	
7	Boundary	7	6	6	6	6	Error	

The typical values in tests 1 and 2 are chosen randomly; in fact they're chosen to make calculating the expected result easier! Test 3 includes two equal highest and lowest scores to check that only one of each is disregarded, and test 4 has a result that is not an integer. The final three tests check the

boundary conditions; this is only tested with Judge 1, so there is an assumption that the other judges' marks are processed in the same way.

Example 2 – Flossy's calculator

Flossy has a simple program that allows her to do some basic mathematics. It allows her to add, subtract, multiply and divide. She has tested this program thoroughly and believes that it works. To help her with some more complicated mathematics she adds a square root function for integer values to her program. What should she add to her test plan?

A range of typical values is easy to find; pretty much any positive values will do. Since there is no square root of a negative number[7], all negative values are error cases, so Flossy will have to be sure that her program works correctly with those, presumably by generating some sort of error message. The program should correctly find the square root of 0 (which is 0), so this is a boundary.

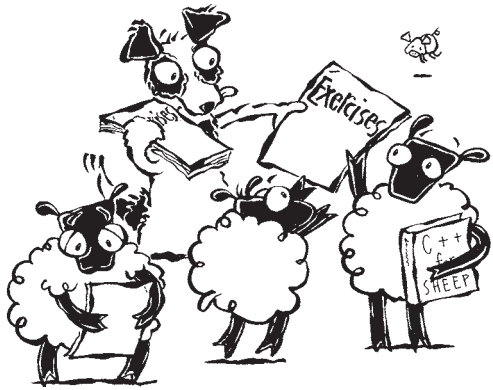
The extra test cases in her plan would be:

	Purpose	Input	Output	Pass
1	Typical	4	2	
2	Typical	15	3.87	
3	Boundary	0	0	
4	Boundary	–1	Error	
5	Boundary	1	1	

This time there are test cases on both sides of the boundary (tests 3 to 5). There are also assumptions that if the program correctly deals with –1 it will do the same with –2 and that an accuracy of two decimal places is sufficient. This accuracy is determined by the application; if the program were to be used in a more complex application, such as airliner navigation, it would be necessary to use and check many more decimal places.

Flossy would also have to run all the other test cases for her program before being confident that her new program worked. It is entirely possible that adding this new function to her calculator has introduced errors elsewhere in the program.

7 Only an imaginary one, of course.



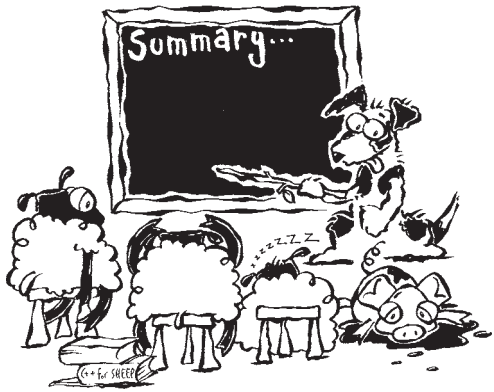
9.1 Type Flossy's Chocolate Sharing program into your computer (or get it from the web site), compile it and execute it. Use the test plan to see if it is correct in all cases. Identify any cases where the program does not work correctly.

9.2 The case of dividing a number by 0 has been identified as a potential error in a program. Write a small program to find out what your C++ system does when faced with a division by 0. You don't know how to stop this happening yet (that's in Chapter 13), but at least you'll recognise it when it happens!

9.3 In the first example in this chapter the assumption was made that if the first judge's marks were handled correctly all the others were also handled correctly. Is this a sensible assumption to have made?

9.4 A program takes the marks achieved by a student on three tests and displays the average. Each test is marked on a scale of 0 to 100 inclusive. Write a test plan for this program.

9.5 The program from Exercise 9.4 is extended to display the class of the student rather than the average. Three classes are defined: A (for average marks 70 to 100), B (average 40 to 69) and C (39 and below). Write a test plan for this version of the program.



Testing is important. As well as having to learn to program you are now going to have to learn to test your programs. It will be best if you get into the habit of developing and using thorough test plans. Never be tempted to treat testing as a random activity. It can be tedious, but it is very important.

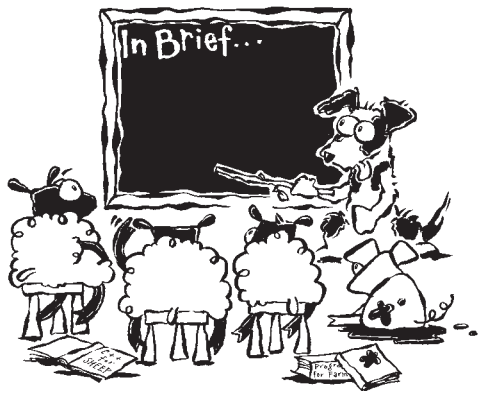
A test plan provides a list of test cases that can be applied to a program. Each test case consists of a set of input values and the corresponding set of output values. A program has passed the test if and only if it passes all the test cases. Only then should a programmer have any confidence in the program. Only then should a programmer be prepared to pass the program on to its intended users.

The test plan should be written before the program. Ideally it should not be written by the programmer or by anyone who has any knowledge of the details of how the program works. The test plan should certainly not be written as an afterthought when the program is believed to work!

Writing a program is only part of the activity of programming. Testing the program and correcting the errors revealed is another, equally important, part.



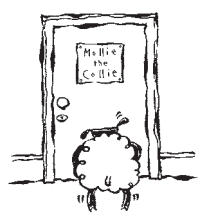

Chapter 10
Defining
C++ Objects
SHEEP
Wool
ear
tail
eye
head
knee
leg
Object
Definition
Public Private
int
char

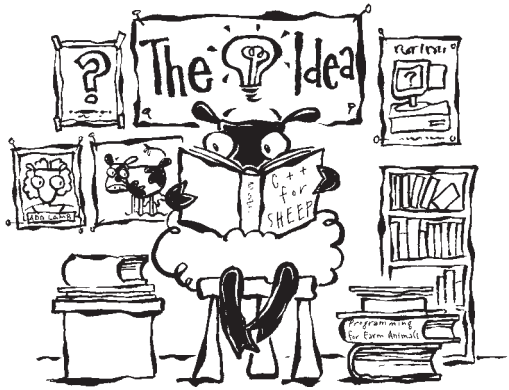


Now it's time to see how object types can be implemented in C++. In a similar way to analysis and design, this is a process that involves two stages; first the attributes and methods of the object type are defined and then the methods are implemented. This chapter is about the first of these two stages (the second stage is in Chapter 12).

In C++ an object type is called a *class*. A class is defined first in a *header file*; this file describes the attributes and methods of the class and provides the *interface* to a class that is later used by a programmer developing a program that uses the class. The methods are implemented only after the class has been designed and specified in the header file; program code to implement the methods is written in the *implementation file*. A programmer writing a program that uses a class doesn't need to know any of the details of what is in the implementation file, and we'll leave this file well alone until Chapter 12.

After reading this chapter you should understand the format of a C++ header file. You should be able to read a header file and understand how the class it defines is intended to work. You should be able to identify a class in a problem area and then specify its attributes and methods in a suitable header file.





Object types in C++ are called *classes*. Classes have all the properties of object types; there are attributes, which hold a single value of a particular type, and methods, which are procedures[1]. When a C++ program is written, the classes it will use are always designed and implemented first. The implementation of a class is a two-stage process; first the class is specified in a *header file* and then the methods are implemented in a separate *implementation file*.

The reason for this split is efficiency. Once the header file has been written, the development of the class and the development of programs that will use the class can happen at the same time. The programmers working on the programs know what the *interface* of the class will be; they know what methods will be available, what these will do and what types of values they process and produce. Moreover, these programmers never need to know exactly how the class is implemented; all they need to see is specified in the header file. This parallel working can greatly shorten the time taken to produce complex systems; it is another advantage claimed for object-oriented programming.

The header file is normally given a `.h` extension, and the name is customarily the name of the class defined in the file:

```
sheep.h
borrower.h
```

Be sure to check your *Local guide* to see whether your system uses the same convention (or find out what the convention is).

Public and private data

A C++ header file defining a class is split into two parts. These two parts specify the *public* and *private* parts of the class. The distinction between these two is straightforward. Anything defined in the public part of the file is available to any program that might make use of the class, while anything defined in the private part is available only to the class itself.

This is a very powerful mechanism. It means that, as long as the public part of the class (the *interface*) remains constant, the private part can be changed if needed. This means that a programmer using the class need only know about the class's public face; the private face remains hidden and can be changed.

---

1 A more correct name for methods in C++ is, arguably, *functions*.

At the same time, the programmer working on the class can change the details of the private part, perhaps to implement something more efficiently, without risking any unpleasant effects on programs that use the class.

An analogy would be the controls of a car. The normal controls – the steering wheel, pedals, handbrake and so on – remain constant and form the public interface to the car's engine. Any changes that a mechanic might make to the engine itself are totally independent of this interface; anything that changes in the engine is private and is something that the driver doesn't need to know anything about. The public interface of the car always works in the same way, and this cannot be altered by any changes made to the engine.

The convention used in this chapter (and throughout this book) will be that attributes are defined in the private part of the class and methods are defined in the public part. This is not required, but it is easily the most common approach. It also allows for a powerful and secure mechanism called *data hiding*, as we will see in Chapter 12.

The format of a header file

A header file follows a fixed format. It starts with the word `class` and the name of the class, then specifies the private part and finally the public part. A template for this in C++ looks like this:

```
class <class name> {
   private:
      // private parts go here
   public:
      // public parts go here
};
```

Braces are used to mark the start and end of the class, but the last line, with the closing brace, is important. This is the only place in C++ where a semi-colon follows a brace! Leaving off the semi-colon will cause many errors[2].

Defining attributes

The first part of the class is the private part, where the attributes of the class are listed. The attributes of a class are defined in a very similar way to the way in which variables are defined. The only difference is that they cannot be given an initial value. The format of the declaration is as expected:

```
<type> <identifier>;
```

For example:

```
string name;
int age;
```

The same guidelines apply to the choice of identifiers for attributes as apply to the choice for variables. The identifier should be short and concise, but should still convey its meaning. The type can be any of the types that are available for variables. In more complicated programs the type can also be

2 As will, incidentally, including a semi-colon after the closing brace used elsewhere in a program!

another class, to allow for the cases that we have seen where an object type has an attribute that is itself another object type.

As an example, here is a class to store information about instances of an object type *Sheep*. The information to be stored about each sheep is the name, the age and the current location in a field; hence these are the attributes of the class. The location is stored as two separate integer variables, representing *x*- and *y*-co-ordinates in the customary way. The header file including the attributes would be:

```
#include <string>

using namespace std;

class Sheep {
  private:
    string name;  // the sheep's name
    int age;      // the sheep's age in years
    int x, y;     // the sheep's position in the field
  public:
    // methods go here
};
```

As in a program, it is necessary to indicate that standard C++ names will be used[3]:

```
using namespace std;
```

and in this case the `string` library must be included, since one of the attributes is of the `string` type:

```
#include <string>
```

The convention used here for naming classes is that the identifier is singular rather than plural[4] and has an initial capital letter. The header file can also include other libraries (such as the `string` library here), which are in fact other header files. It is also often useful to include clear comments to indicate the meaning of each attribute. Admittedly, it is not strictly necessary in such a simple example as this, but it is a good habit to get into.

Declaring methods

When analysing a problem, three pieces of information were determined about each method required for an object type. Not surprisingly, the same three pieces of information are required to specify a method in a header file. These are the name of the method, the type of the value that it determines, and the types of any values that it processes.

The format is not unlike the declaration of a variable:

<type determined> <name of method> (<types of values processed>);

This definition is sometimes called the *prototype* of the method. For example:

```
bool moveTo (int, int);
```

3 Namespaces really are complicated. Eventually you will come to realise that this is not always the best idea, but including it in every header file is safe for now.

4 Sheep is, of course, both singular and plural. *Sheeps* would be wrong.

The types of values processed by the method are specified in a comma-separated list; if there are no values this list is just left blank (but the brackets remain). The values inside the brackets are called the *parameters* of the method, or sometimes the *arguments*. A method may automatically make use of any of the values in the private part of the class (and also any of the other methods); the parameters are used to provide additional values. The type of value that the method determines is called the *return type* of the method.

Some methods do not return a value, or more accurately return a value that is nothing. This explains the need for a data type, `void`, that indicates this. A method that does not return a value is defined as returning a `void` value.

A definition of a method contains a lot of information. This information is vital for the programmer who will be writing programs that will make use of the class and its methods. A programmer must be able to read a header file and understand precisely what it conveys about the interface of the class.

Here are some examples:

```
int getAge ();
```

This is a method called *`getAge`*. It returns an integer value and does not require any parameters. Some programmers might prefer to specify that no parameters are required explicitly by writing:

```
int getAge (void);
```

but this is not usual.

```
void printNeatly ();
```

This method, called *`printNeatly`*, does not require any parameters and does not return a value.

```
bool atPosition (int, int);
```

This method does require parameters: two integers. It is called *`atPosition`* and it returns a Boolean value.

The methods that a class would have would depend, of course, on the needs of the programs that would use the class. In the current example, the *`Sheep`* class will be extended with a few possibilities:

- a method to do nothing more than return the sheep's age
- a method to display the sheep's name and position neatly
- a method to determine how far away the sheep is from a particular position (this position would be provided as two parameters)
- a method to return true if a sheep is at a particular position (again provided as two parameters)

The header file now becomes:

```
#include <string>

using namespace std;

class Sheep {
  private:
    string name;  // the sheep's name
    int age;      // the sheep's age in years
    int x, y;     // the sheep's position in the field
```

```
  public:
    int getAge ();        // return the sheep's age
    void printNeatly (); // print all details of a sheep
    float distanceFrom (int, int);  // return distance from
                                    // a position
    bool atPosition (int, int);   // true if sheep is at
                                  // position provided
};
```

Constant methods

Some methods change (or might change) the values of the private attributes. Some, on the other hand, do not. C++ provides a mechanism for the designer of a class to indicate to the programmer that a particular method will never change any of the values. This also means that a programmer implementing the class itself is less likely to make a mistake by changing these values accidentally – the compiler will report an error if such an accidental change is detected.

The information that using a method will not change any of the private values is added simply by appending the word `const` to the end of the method definition, before the semi-colon. In the example the method that just displays the details of a sheep should obviously not change any of the values of the sheep; it should therefore be written:

```
void printNeatly () const;
```

In fact, in the example none of the methods would change any of the attribute's values, so all four should have `const` appended:

```
public:
  int getAge () const;
  void printNeatly () const;
  float distanceFrom (int, int) const;
  bool atPosition (int, int) const;
```

This adds one final piece of information to the method's specification.

There remains one final thing to add to the class. A special type of method to create an instance of the class is needed.

Declaring objects and the constructor

Each class should also have one special method, called a *constructor*. This method is responsible for creating the instance of a class when one is used in a program. It follows that this method must be declared in the public part of the class. An object is declared in a program in the same way as a variable, so a *Sheep* object would be declared simply:

```
Sheep flossy;
```

This declaration implicitly calls the constructor. Internally the constructor is responsible for allocating sufficient of the computer's memory to store the attributes of the object but, happily for the programmer, all of this takes place automatically.

The constructor must have the same name as the class and does not usually take any parameters[5]. The declaration is just:

---

5 Actually, a constructor can take parameters. An obvious possibility is some initial values for the attributes of the class. If this is required, the prototype has the types of the parameters specified in the brackets, as you might expect.

<class name> ();

For example:

```
Sheep (); // Constructor
```

The addition of a constructor gives the final header file for the *Sheep* class:

```
#include <string>

using namespace std;

class Sheep {
  private:
    string name;  // the sheep's name
    int age;      // the sheep's age in years
    int x, y;     // the sheep's position in the field

  public:
    Sheep ();     // constructor

    int getAge () const;       // return the sheep's age
    void printNeatly () const; // print all details of a sheep
    float distanceFrom (int, int) const;  // return distance from
                                          // a point
    bool atPosition (int, int) const; // true if sheep is at
                                      // position provided
};
```

The header file

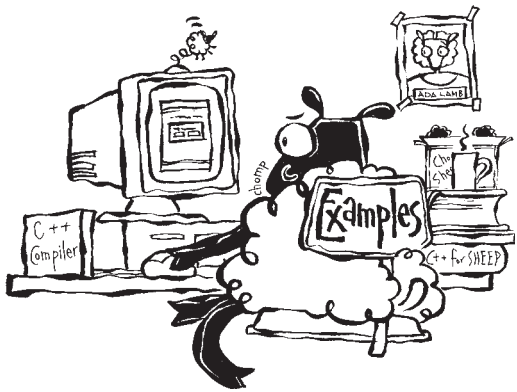
This completed header file provides all the information that a programmer would need in order to write a program that made use of the class. The programmer has no need to know how any of these methods are implemented; all the information is in this single header file.

The programmer will pay most attention to the public part of the file. This is the part of the specification that describes the public face of the class, the part of the class that programs are allowed to make use of – the *interface* of the class. Armed with this, and only with this, the programmer can get to work on writing programs that make use of the class.

The header file for a class must be included in any such programs (or other header files) that make use of it. We have already seen how to include a system library, and the syntax for including a header file is very similar:

```
#include "headerFile.h"
```

The difference is that double quotes are used instead of < and >; this indicates that the file to be included has been written by the programmer and is not part of the C++ system. Normally this also means that the file is stored in a different place on the computer system, usually in the same directory as the implementation of the class. It is usual to list the names of these header files below the included system libraries (my style is also to leave a blank line between the two).



Example 1 – Mollie's library revisited

Mollie's library continues to do good business. Mollie is still developing a C++ program to keep track of her books, and her first step is to develop a class for her `Borrower` *object type. For each borrower she wants to be able to store the name and the unique borrower number that she has assigned to each one. The first program that she intends to write does nothing more than print a neatly formatted list of all her borrowers.*

What should the header file for the `Borrower` *class be?*

There are two attributes, the name and borrower number. The first is clearly a string (so the class will need to include the `string` library), and the second is presumably an integer[6]. These two attributes will form the private part of the class.

Mollie's list of all her borrowers will need to access this private information, so a method to access them will be needed in the public part. This will simply take these two values and print them out in some neat way; it will not return a value and does not need any additional values as parameters. The only other method will be the constructor.

The header file would be:

```
#include <string>

using namespace std;

class Borrower {
  private:
    string name;
    int number;

  public:
    Borrower ();

    void print () const;
};
```

6 If there was any doubt about this in a real application, the correct approach would, of course, be to seek clarification before starting to design the class.

Example 2 – Mollie extends the program

The next step in the development of Mollie's program is to record which books have been borrowed by which borrowers. Mollie needs to add a `Book` *class. For each book she wants to store the name, author, the identity of the borrower and the date the book is due to be returned.*

What should the header file for the `Book` *class be?*

Once again the attributes are straightforward to identify. The name and the author are both strings and the borrower is an instance of the *`Borrower`* class (so the header file defining this class will need to be included). The date remains a complication, and it would probably be a good idea if Mollie implemented a class for that too.

There would again be two methods: one to record that a book had been borrowed and one to record a return[7]. The first of these would require the identity of the borrower as a parameter. Finally, there would of course also be a constructor.

Assuming that there is a *`Date`* class available[8] the header file would be:

```
#include <string>

#include "borrower.h"
#include "date.h"

using namespace std;

class Book {
  private:
    string title;
    string author;
    Borrower onLoanTo;
    Date dateDue;

  public:
    Book ();

    void borrow (Borrower);
    void bringBack ();
};
```

Example 3 – Mollie and dates

Mollie decides that a class to store dates is exactly what she needs. Not unreasonably, she suspects that someone will have written such a class before, and she soon finds one that Flossy has written for a different program.

The header file for this class is:

```
using namespace std;

class Date {
  private:
    int d, m, y;
```

7 It would record a return but it cannot be called simply "return". `return` is a reserved word in C++ and using it as an identifier for an attribute or method is an error.

8 The header file defining this class will also have to be included, of course.

```
  public:
    Date ();

    bool isValid () const;
    void next ();
    void advance (int);
    void print () const;
    int daysBetween (Date) const;
};
```

Sadly Flossy has forgotten to put any comments in the file. What does the header file tell Mollie about the class?

The file tells Mollie just about everything that she needs to know in order to use the class; the only problem is that the absence of comments means that she will have to rely on some educated guesswork.

She will pay most attention to the public part of the class, but a quick glance at the private part confirms that Flossy has used the obvious strategy to store dates; they are stored as three separate integers: one each for day, month and year.

The public part is much more interesting. The first entry is the constructor and then there are five potentially useful methods. Taking them one at a time:

```
bool isValid () const;
```

This method presumably determines whether or not a given date as stored in the three private attributes is valid. The obvious implementation would be to return true if the date were valid and false otherwise. This would be useful when checking that a date entered was valid, so that, for example, 31 September would be rejected. Hopefully it would also handle the complications of leap years and 29 February.

```
void next ();
```

The most likely explanation for this method is that it moves the date on to the next day. It would include some logic to deal with the varying numbers of days in months and leap years and so on, but the complexities of how it works are not important to a programmer using the class. This method does not return an interesting value, but the absence of a `const` confirms that it does indeed change the values of the attributes.

```
void advance (int);
```

This method takes a single integer parameter, returns no value, and changes the values of the attributes. The most likely suggestion is that it advances the date by the provided number of days, quite probably by calling the *next* method several times. Once again there is no `const` on this line, which confirms that the method changes the values of the private attributes.

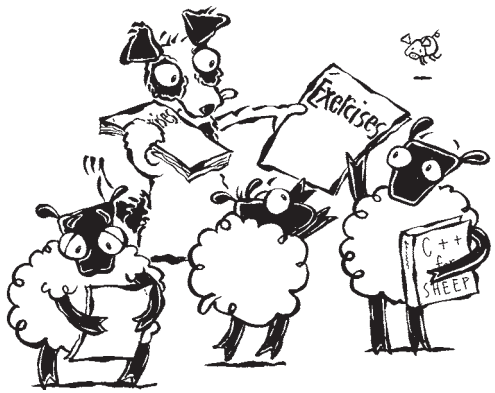
```
void print () const;
```

The purpose of this method is fairly obvious. It simply prints the date in some useful format; it achieves this without changing any of the values. Unfortunately, the absence of any comments means that Mollie can only guess as to what the format might be.

```
int daysBetween (Date) const;
```

Finally, this method takes as a parameter another instance of the same class. From its name and the return type it seems most likely that it calculates and returns the number of days between two dates.

It is possible to extract a lot of information from this file, but the absence of comments is a problem. Assuming that Flossy has used sensible names for all the methods, the suggestions about the purpose of each method are probably correct, but it is not possible for Mollie to be sure. There is a lot of information that can be extracted with some precision from the definition of each method, but a comment is always needed to confirm the purpose of a method. In practice, Mollie would have to write a set of small programs using the class to be sure of what each method does.



10.1 Take the header file from the third example in this chapter and add suitable comments to the file.

10.2 Identify and explain the errors and omissions in the following header file.

```
using namespace std;

class Pig {
  private:
    string name;
    int age;
  public:
    Pig ();
    getAge ();
    bool tooOld () const;
    bool olderThan (Pig);
}
```

10.3 Identify and explain the errors and omissions in this header file:

```
using namespace std;

class Cow {
  private:
    Cow ();
    float weight;
    int age;
```

```
  public:
    bool tooOld () const;
    bool olderThan (Pig);
};
```

10.4 Examine the following definition for a *Farmer* object type. Write out the header file that would be used if this class were implemented in C++.

Farmer object				
Attributes				
Identifier	**Description**		**Type**	
name	The farmer's name		string	
address	Home address of the farm		string	
telephone	Phone number		string	
current	True if the farmer is a current customer		Boolean	
animals	Number of animals bought this year		integer	
value	Value of animals bought this year		floating-point	
Methods				
Identifier	**Description**	**Returns**	**Return type**	**Parameters**
changePhone	Updates phone number	Nothing	Null	None
delete	Sets Current attribute to false	Nothing	Null	None
addSale	Adds details of a new sale, adding to Animals and value	Nothing	Null	Animal object
totalValue	Determines the total sales made to this farmer	Total sales	Floating-point	None
averageSale	Determines the average value of a sale	Total sales – total value divided by number of animals	Floating-point	None
betterCustomer	Determines whether farmer has spent more than another	True if this farmer has spent more	Boolean	Farmer object

10.5 Examine the following header file (the file name is *farmhouse.h*). Write out a description for the object type that is implemented in this class. Take a copy of the file and add suitable comments.

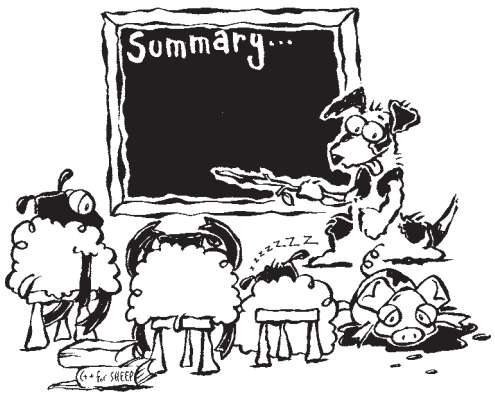
```
#include <string>

#include "farmer.h"

using namespace std;

class FarmHouse {
  private:
    string address;
    Farmer owner;
    bool doubleGlazed;
  public:
    FarmHouse ();
```

```
    void printAddress () const;
    void changeOwner (Farmer);
    bool hasDoubleGlazing () const;
    void installDoubleGlazing ();
};
```



Objects are implemented in C++ as classes. A class is defined into two files: a header file and an implementation file. The header file provides a full specification of all the attributes and methods and describes which parts of the class are public and which are private. The private part of a class cannot be used in any programs that make use of the class; only the public part of the class is available.

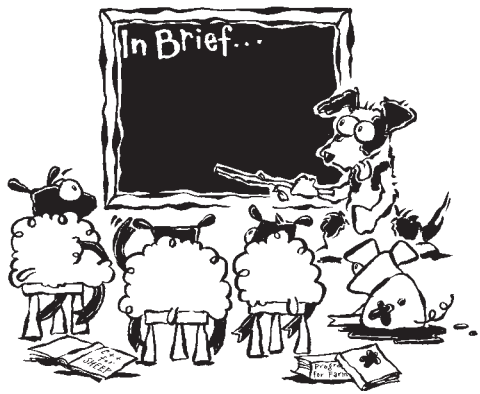
The header file follows a fixed format (and includes the only occasion in C++ when a semi-colon immediately follows a closing brace). After the name of the class is defined, the attributes are listed in the private part followed by the methods in the public part. Attributes are always defined in the private part and methods are always defined in the public part.

The definition of attributes follows the same format at the definition of C++ variables. The definition of methods has a similar format; methods may also have arguments or parameters, which are values that are not part of the class but which are needed by the method.

An advantage of this separation of the implementation into two files is that it is possible for a programmer to work on a program that uses a class with only the header file. The header file describes the interface to the class; this is all the information the programmer needs.

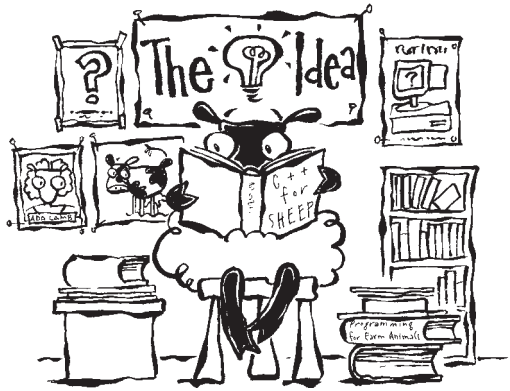
The next stage in learning to write C++ programs is to learn to write programs that make use of classes. We can do this before we know how to implement a class, so it's what we'll do next.


Chapter 11
Using C++
Objects
Methods
Baa!
Eat



After reading the last chapter you should now be able to read and understand a C++ header file that defines a class. This means that you are now able to write simple C++ programs that make use of classes written by other programmers. You can in fact do this before you can write classes yourself; all the information that you need is provided by the interface specified in the header file. This is, of course, one of the advantages of object-oriented programming.

This chapter takes the basic C++ statements that you have learned up to this point and shows you how the programming ideas that you have used and practised so far can be extended to produce C++ programs that make use of objects. After reading this chapter you will understand how to create and use objects in a C++ program. You should be able to read a header file, understand it and make use of the class that the file describes in your own programs.



A C++ header file describes, among other things, the *public interface* of the class. This part of the header file is the important one in this chapter; you will need to be able to examine the public part of a class and so work out how to make use of the class in your programs. You will normally be able to see the private part of the class too, but you will not be able to directly access any of the values that you see there.

To recap, the basic format of a header file is:

```
class <ClassName> { // the name of the class
  private:
    // your programs cannot use this part of the class
    // this contains the attributes of the class
  public:
    // your programs can use only this part of the class
    // this contains the methods of the class
};
```

The public part of the class normally specifies the methods that can be used on instances of the class (objects). It may also specify values or attributes, but this is unlikely and is usually bad programming practice. The specification of a method tells a programmer using the class the name of the method, the type of value it returns, and the types of any values that it requires as parameters. Anything that is not clear in the header file should always be explained by suitable comments.

Preparing to use a class

A program that uses a class is said to *call* the methods of the class. It might be called the *calling program* of the methods.

Before using a class there are two preliminary mechanical steps that you must take. The first is to include the definition of the class in your program; this is very similar to the way in which a header file defining one class had to include the definitions of any other classes that it used. As before, this is done with a `#include` statement similar to the ones that you have used before to include various system libraries such as `iostream`. The format in this case is subtly different from that used for system libraries:

#include "<*classfile*>"

Here *`classfile`* is the name of the file on your computer that contains the header file. It is almost always best to store it in the same directory as your program, but this is not essential. It is customary and good practice for the name of the file to also be the name of the class, but this is not enforced; the compiler that processes this statement will infer nothing from the name of the file.

The difference between this `include` statement and those used for system libraries is that quotation marks are used rather than angled brackets to enclose the file name. Contrast the two:

```
#include <iostream>
#include "classfile.h"
```

The rule is simple: use angled brackets for system libraries and quotation marks for files written by you or other programmers. Header files are the only files that should ever be included this way; unfortunately, including an implementation file or a program in this way will probably not generate an error from the compiler, but it certainly is an error and will cause all sorts of problems. Never do it!

The second mechanical detail is that you now need to be able to compile programs that are stored in more than one file. One file will be the program that you write yourself and the other will be the implementation file of the class. While you don't need to know what's in this second file, you do need a copy of it, probably in the same directory as your program. As you've probably come to expect by now, the way in which you compile programs that use

a class's implementation file is different on different C++ systems. As always, I'll have to refer you to your *Local guide* to provide the details you need.

Declaring an object

The first stage in using an object (an instance of the object type defined in your class) is obviously to declare one. The format of the declaration is exactly the same as that of any other declaration, and the same as the format for declaring an attribute that is itself a class. You specify the type of the instance to be declared and provide an identifier:

<class name> <identifier>;

The name of the class should be written exactly as given in the header file; this includes the case of the letters. For example, if a header file declares a class:

```
class Sheep {
```

the correct declaration of an instance of the class (a *Sheep* object) is:

```
Sheep flossy;
```

and not:

```
sheep flossy;
```

This declaration actually implicitly calls the constructor of the class. This is the method that is responsible for allocating the memory that will store the class instance. The constructor may also allocate default values for some of the attributes; there should be comments in the header file to warn you (and explain the values that are used) if this is the case.

A class instance declared in this way is effectively the same thing as the variables you have used before. The effect of defining a new class has been to add a new type for variables to those available.

As with any variable, more than one class instance can be declared in a single declaration:

```
Sheep flossy, freda;
```

Of course it makes no sense to have a form of the declaration that allows the programmer to assign an initial value to each instance (any initial values are set by the constructor). It does make sense to check the comments to see whether or not an initial value is provided, and it might make sense to add a comment to the declaration to indicate what these are.

Calling a method

A method is called on a class instance in a statement of the form:

<class instance>.<method name> (<parameters>);

This is not as complicated as it looks! The identifier of the class instance has the name of the method appended to it with a full stop between. The parameters are then specified in brackets after the method name. If there are several parameters they are separated by commas – this time the brackets contain the identifiers of the parameters and never their types. If there are no parameters the brackets must still be present, but there is nothing between them.

An example should make this clear. The *Sheep* class from the previous chapter defined this set of methods:

```
int getAge () const; // return the sheep's age
void printNeatly () const; // print all details of a sheep
float distanceFrom (int, int) const;  // return distance from
                                      // a point
bool atPosition (int, int) const;   // true if sheep is at
                                    // position provided
```

Assuming that the program had declared:

```
Sheep flossy;
```

the statement to call the *printNeatly* method to display Flossy's details would be:

```
flossy.printNeatly ();
```

This method has the return type `void`, indicating that it does not return a value. There are no parameters, so the brackets are empty.

The other three methods, on the other hand, do return values, and the final two have parameters.

The return values are presumably of interest to the programmer using the class, so they should be stored in a suitably declared variable of the appropriate type. This gives a slightly longer form of a call to a method:

<variable> = <class instance>.<method name> (<parameters>);

Another example – to use the *getAge* method to retrieve Flossy's age and store it in a suitable program variable the statement is:

```
int age;
age = flossy.getAge ();
```

This method has no parameters, so the brackets are present but, as usual, contain nothing. This assignment could also have been combined with the declaration of the program variable to provide an initial value for that variable:

```
int age = flossy.getAge ();
```

If a method requires parameters the values are included in the brackets. If there is more than one value they must be provided in the correct order (the types of the parameters must match the types specified in the header file). The *distanceFrom* method returns the distance that a sheep is from a particular point. To find out how far Flossy is from the point (5, 3) and store that distance in a program variable called *away* the statement would be:

```
float away = flossy.distanceFrom (5, 3);
```

This is different from:

```
float away = flossy.distanceFrom (3, 5);
```

which refers to the point (3, 5). The correct number of parameters must be provided in the order specified in the header file.

This call to the method would result in a compilation error:

```
float away = flossy.distanceFrom (4); // Error!
```

since the number of parameters does not match the number specified in the header file.

Assignment

Classes are obviously very similar to the standard data types that you have used in your programs so far. They are used in much the same way and much of the syntax used with them is the same as is used with, say, integers. The important difference to remember is that the header file defines absolutely everything that your programs can do with an instance of the class. The general rule is that if it's not defined in the header file, and specifically in the public part of the header file, there's no way to do it in a program.

There is one special case, however. It is often very useful to be able to assign the values of all the attributes of one instance of a class to another. To provide this, all classes have a default *assignment operator* which does just this. It works as expected:

```
Sheep flossy, freda;
flossy = freda;
```

This statement would assign the values of all the attributes in the instance *`freda`* to the corresponding values of *`flossy`*.

This is the only operator that is defined by default[1].

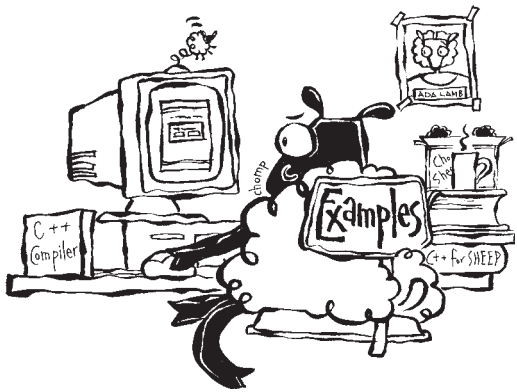
Using a class

A class effectively extends the range of data types that you have at your disposal as you write your programs. The public interface in the header file gives you all the information you need in order to use the class. After "including" the header file in a program, class instances are declared in a similar way to variables.

Methods are called on an instance of a class by joining the name of the method and the identifier of the instance with a full stop. If a method requires parameters, the correct number must be provided in the expected order and must match the expected types.

This really is not as complicated as it might seem! Now it's time for some examples of how this all works.

1 Operators can be defined as part of the class, but that's a bit beyond what we need at the moment.



Example 1 – A simple sheep class

Farmer Strawstrap likes to keep track of where his sheep are. He gets very concerned if they stray too far away from the farm; if they do he has to send Mollie to fetch them. He wants a small C++ program that helps him keep track of the locations. Mollie has developed a class to store details of the sheep. The header file is:

```
// sheep.h
//
// Class to store details and locations of sheep
//
// Author: Mollie
// Date:   22nd November 2001
// Platform: g++, Linux

#include <string>

using namespace std;

class Sheep {
  private:
    int x, y; // position
    string name;
  public:
    Sheep ();

    void moveTo (int, int); // move sheep to a point
    void setName (string);  // sets the sheep's name
    void printLocation () const;
    void printNameAndLocation () const;
    void printIfTooFarAway () const;
          // prints warning if sheep is too far away
    float distanceFrom (int, int) const;
  };
```

Write a program that declares two instances of the class and then prompts the user to enter their positions and names. Use the `printNameAndLocation` method to print the positions neatly and the `printIfTooFarAway` method to display a warning if either sheep has strayed.

You can find an implementation file for this on the web site.

The first step in developing this program is to add the line to include the file containing the definition of the class to the standard template for any

program. The header block of comments reveals that the file is called *sheep.h*, so adding this to a handy template file provides a program that starts as:

```
#include <iostream>

#include "sheep.h"

using namespace std;

int main ()
{
  return 0;
}
```

It would be good practice to compile this program even at this early stage, just to check that all is well. A missing bracket is much easier to spot and correct at this point.

The declaration of the sheep object instances is simply:

```
Sheep sheepOne, sheepTwo;
```

A simple dialogue can be used to get the user to enter the names and initial positions of the sheep. Some temporary variables are needed to store these values before they are saved in the object instances; there is no way to access the attribute values themselves, as they are defined in the private part of the class. For the first sheep the dialogue might be:

```
int x, y;
string name;

cout << "Enter the first sheep\'s name: ";
cin >> name;
cout << "Enter " << name << "\'s x position: ";
cin >> x;
cout << "Enter " << name << "\'s y position: ";
cin >> y;
```

These values are then copied into *sheepOne* using the methods:

```
sheepOne.setName (name);
sheepOne.moveTo (x, y);
```

The dialogue for the second sheep is the same and can use the same temporary variables, since their values for the first sheep have been safely saved away.

```
cout << "Enter the second sheep\'s name: ";
cin >> name;
cout << "Enter " << name << "\'s x position: ";
cin >> x;
cout << "Enter " << name << "\'s y position: ";
cin >> y;

sheepTwo.setName (name);
sheepOne.moveTo (x, y);
```

Once the values have been entered and saved the locations of the two sheep can be printed:

```
sheepOne.printNameAndLocation ();
sheepTwo.printNameAndLocation ();
```

and a similar call to the *printIfTooFarAway* method will display a warning message if needed.

The complete program would be:

```
// wanderingSheep.cc
//
// Checks whether sheep have wandered worryingly far
//   away.
// Chapter 11, Examples 1.
//
// Author:   AMJ
// Date:     25nd November 2001
// Platform: g++, Linux

#include <iostream>

#include <string>

#include "sheep.h"

using namespace std;

int main ()
{
  Sheep sheepOne, sheepTwo;

  int x, y;
  string name;

  cout << "Enter the first sheep\'s name: ";
  cin >> name;
  cout << "Enter " << name << "\'s x position: ";
  cin >> x;
  cout << "Enter " << name << "\'s y position: ";
  cin >> y;

  sheepOne.setName (name);
  sheepOne.moveTo (x, y);

  cout << "Enter the second sheep\'s name: ";
  cin >> name;
  cout << "Enter " << name << "\'s x position: ";
  cin >> x;
  cout << "Enter " << name << "\'s y position: ";
  cin >> y;

  sheepTwo.setName (name);
  sheepTwo.moveTo (x, y);

  sheepOne.printNameAndLocation ();
  sheepTwo.printNameAndLocation ();

  sheepOne.printIfTooFarAway ();
  sheepTwo.printIfTooFarAway ();

  return 0;
}
```

This program highlights another advantage of using a class. The program does quite a lot (and behind the scenes some of it is quite complicated[2]) but it is still reasonably short; a lot of the processing is taking place in the implementation of the methods. Also, you can use methods that use programming techniques that you yourself have not learned (this class does – take a look). The method to print a warning if a sheep moves too far away certainly uses C++ that you have not seen yet.

You can find an implementation of the class on the web site. This program does not use all of the methods defined in this class; this is quite normal. The class will be developed so that it can hopefully be used in many programs and it is far from likely that any one program will need to use all the methods.

Example 2 – Pigs have classes too

Farmer Strawstrap also uses programs to keep track of his pigs. He is especially interested in the value of each pig, something that is determined by a range of factors combined in incredibly complex formulas. He wants a program that will take the details of two pigs and will calculate and display the difference in value between the two.

Mollie has developed a C++ class to store details of pigs. The public interface is:

```
public:
  Pig ();

  void print () const;
  void setName (string);
  void moveTo (int, int);
  void setFactors (int, int, float);
    // parameters are age, height (cm), weight (kg)
    // determines and sets the value
  float getValue () const; // returns the value
  float valueDifference (Pig) const;
    // finds value difference between two pigs
};
```

Using the information in this interface write the program that the farmer requires.

You can find an implementation of this class with the sheep class on the web site.

The program follows a pattern similar to the one in the previous exercise. The mechanical step is much the same; the header file, presumably *pig.h*, is included at the top of the program. The two required object instances (*pigOne* and *pigTwo*) are declared in the usual way.

A similar dialogue would be used to get the values for the three relevant factors from the user and store them in temporary variables. There would then be a method call to *setFactors* to set the attributes and determine the value, all done without our knowing anything about the incredibly complex formulas being used[3]. It seems very likely that these factors correspond to three attributes in the private part of the class, but there is no way of knowing this for certain. In any event, this internal detail is quite unimportant to someone using the class; all the information needed is in the public part of

2 Look at the implementation of the `distanceFrom` method, for example.

3 If you take a look, the formulas are not in fact *that* complex.

the definition. Care would be need when using *setFactors* to make sure that the parameters are supplied in the expected order; this order is given in the comment and is age, then height, then weight.

Similarly, it seems likely that the value is stored as an attribute (it isn't, as it happens!), but again there is no way to be sure. What matters is that the *getValue* method returns the value and that the *valueDifference* method has access to the correct figures. In fact, for this program only the second of these is needed (although we might expect that *valueDifference* makes use of *getValue* (it does!); the difference in value can be printed simply:

```
cout << "The difference in value is "
     << pigOne.valueDifference (pigTwo) << endl;
```

Of course this happens to be identical to:

```
cout << "The difference in value is "
     << pigTwo.valueDifference (pigOne) << endl;
```

It would not matter which was used.

Again there are many methods and possibly many attributes in this class that are not used. Also there are many details of the implementation of the class that a programmer using it needs to know nothing about. Both these things are quite usual.

The final program would be:

```
// valuablePigs.cc
//
// Determines relative values of two pigs.
// Chapter 11, Examples 2.
//
// Author: AMJ
// Date: 25th November 2001
// Platform: g++, Linux

#include <iostream>

#include "pig.h"

using namespace std;

int main ()
{
  Pig pigOne, pigTwo;

  int age, height;
  float weight;

  cout << "Enter the age of the first pig: ";
  cin >> age;
  cout << "Enter the height of the first pig: ";
  cin >> height;
  cout << "Enter the weight of the first pig: ";
  cin >> weight;

  pigOne.setFactors (age, height, weight);

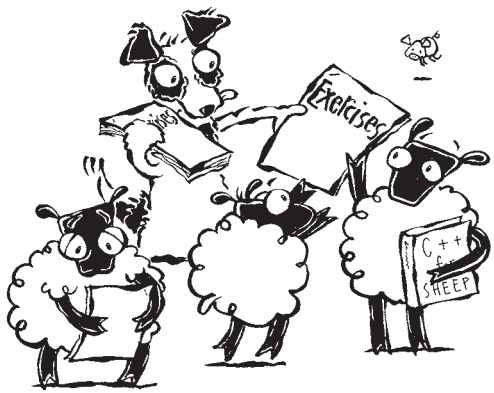
  cout << "Enter the age of the second pig: ";
  cin >> age;
```

```
    cout << "Enter the height of the second pig: ";
    cin >> height;
    cout << "Enter the weight of the second pig: ";
    cin >> weight;

    pigTwo.setFactors (age, height, weight);

    cout << "The difference in value is "
         << pigOne.valueDifference (pigTwo) << endl;

    return 0;
  }
```



11.1 Write a program using the *Pig* class from the second example that will prompt the user to enter the details of a pig and will print the value of that pig. The value should be calculated according to Farmer Strawstrap's incredibly complicated formulas. Remember that you can find the implementation of the class on the web site.

11.2 Examine the following definition for a class to store details of sheep.

```
#include <string>

using namespace std;

class Sheep {
  private:
    string name;
    int age;
    float value;
    int x, y;

  public:
    Sheep ();

    void moveTo (int, int);
    float distanceFrom (int, int);

    bool veryValuable () const;
};
```

Identify and explain the errors in the following program that is intended to make use of this class. Are all the errors in the program itself or is the class inadequate in some way?

```
#include <iostream>

using namespace std;

const int FarmX = 5;
const int FarmY = 3;

int main ()
{
  sheep flossy;

  name = "flossy";

  cout << "Enter the value of Flossy: ";
  cin >> flossy.value;

  cout << "Flossy is "
       << flossy.distanceFrom (FarmX, FarmY)
       << " away from the farm" << endl;

  return 0;

}
```

11.3 Examine the following public interface to a class to store details of cows. Write down a full description of the methods that are available for this class. What do you imagine is in the private part of this class? Does it matter that you don't have access to it?

```
public:
  Cow ();

  void setName (string);
  void setAge (int);
  void setWeight (float);

  string getName () const;
  int getAge () const;
  float getWeight () const;
  float getValue () const;

  void moveTo (int, int);           // moves to x, y
  bool atLocation (int, int) const; // true if at x, y

  float distanceFrom (int, int) const;

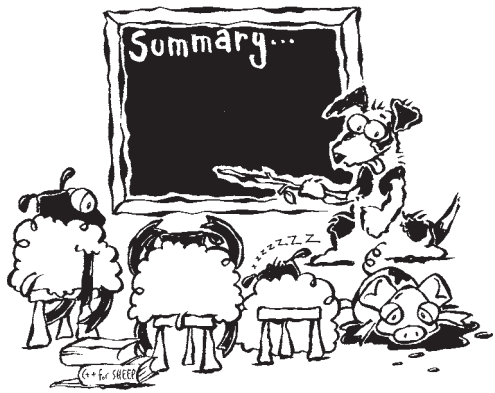
};
```

11.4 Write a program that uses the *Cow* class above. The program should prompt the user to enter the name, age and weight of a cow and should print the value of the cow in a message of the form:

<Cow Name> is worth <Value>.

Assume that the `Cow` class is defined in the file `cow.h`. You can find an implementation of the class in the usual place on the web site.

11.5 Write another program that could be used to keep track of cows. The program should prompt the user to enter a cow's name and current position. The program should then display the distance the cow is away from the milking shed, which you might guess by now is at position (5, 3).



This chapter has explained how to make use of C++ classes in your programs. The classes used have been quite simple, but the basic ideas and syntax remain the same for all more complicated classes. By making use of classes your programs are now able to achieve quite complex results.

The process for using a class is straightforward. The name of the file containing the definition of the class is included in the program (with quotes, not angled brackets) just below the lines including the various system libraries. This done, the class can be used and instances (called objects) can be declared. The syntax for declaring objects is very similar to that for declaring variables of the basic types such as integers and strings.

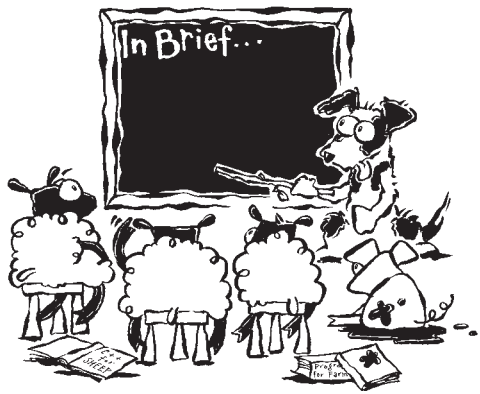
The operations that can be carried out on objects are defined in the public interface part of the class's header file. A programmer using the class has no need to know anything about the private parts of the class or indeed anything about how the methods are implemented. All the information that is needed is conveyed succinctly, accurately and unambiguously in the header file. Where there is any possible ambiguity this should be resolved with comments.

A particular advantage that you have seen of this approach in this chapter has been that you have been able to make use of methods that use C++ statements and techniques that you have not yet learned.

You should now also have learned how to compile the file containing your program together with a file containing the implementation of the class used in the program. Of course, most programs will make use of many classes, but the ideas and the mechanical details are all the same.

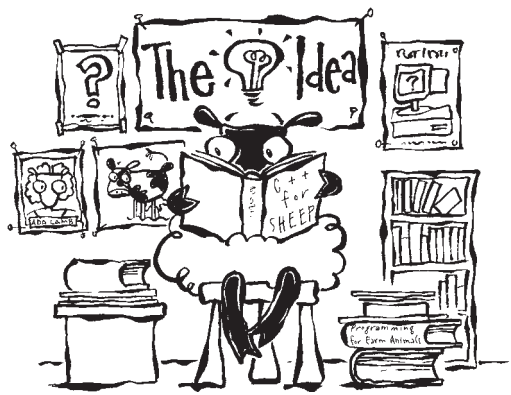
Now that you are able to make use of simple classes it is time to move on to look in more detail at how the methods themselves are implemented. It is time to take a look at what precisely is going on in that mysterious implementation file.


Chapter 12
Get Your Hands off My Data!
GO AWAY!
Public Interface
Private Ahhhubby



By now you should have had plenty of practice at writing simple C++ programs. Some of these programs will have made use of classes developed by other programmers and you should have become accustomed to using the two files that together define a C++ class. The header file provides the public interface to the class and the implementation file provides the methods themselves. The time has come when you should learn how to construct the implementation file yourself. Soon you will be able to develop your own classes for other programmers to use.

After reading this chapter you should understand how to implement simple methods for C++ classes, and so should be able to implement simple classes. You should also understand the concept of *data hiding*, and you should understand how the separation of C++ classes into public and private parts allows programmers to implement this powerful mechanism.



The time has come for you to learn how to write the implementation files of classes. The first stage is to deal with the simplest and most common types of method. This chapter introduces you to three very common and very basic types of method in C++:

- *constructor* – the method that actually creates an instance of a class when it is declared
- *selector* – a method that allows access to a private value from within the class by returning its value (sometimes called an *accessor*)

- *mutator* – a method that allows access to a private value of the class so that that value can be changed

These three types of method will be found in almost every class that is ever written; every class must have a constructor and it is hard to imagine a useful class that did not allow any form of access to its private data. Each of the classes in the previous chapter had these methods, and if you looked at the implementations of the classes in the exercises and examples you will have seen some of each. There is nothing new here.

What is new is that you will now learn how to implement them.

The following sections assume that a simple (and, by now, fairly familiar) *Sheep* class is being implemented. The header file for this class is:

```
// sheep.h
//
// Definition of a simple Sheep class.
//
// Illustrates a simple constructor, selectors and mutators.
//
// Chapter 12.
//
// Author:   AMJ
// Date:     27th November 2001
// Platform: g++, Linux

#include <string>

using namespace std;

class Sheep {
  private:
    string name;  // the sheep's name
    int age;      // the sheep's age in years
    int x, y;     // the sheep's position in the field
    float value;  // the sheep's value in sterling

  public:
    // Constructor
    Sheep ();

    // Selectors
    string getName () const;
    int getAge () const;
    int getX () const;
    int getY () const;
    float getValue () const;

    // Mutators
    void setName (string);
    void setAge (int);
    void setX (int);
    void setY (int);
    void setValue (float);
};
```

This example uses a simple naming convention for the selectors (*get* followed by the name of the value they return) and mutators (*set* followed by the value they affect), and this is probably a good pattern to follow. Obviously each

selector must return a value of the same type as the private attribute they select. The single parameter of each mutator is the value that the private member is to be set to.

A program might use these methods as follows:

```
Sheep flossy;

flossy.setName ("Flossy");  // Set name to Flossy
flossy.setAge (10);         // Set age to 10
flossy.setX (5);            // Set X ...
flossy.setY (3);            // ... and Y co-ordinates

// Now print the values

cout << flossy.getName () << " is "
     << flossy.getAge () " years old and is now at ("
     << flossy.getX () << ", " << flossy.getY () << ')' << endl;
```

This way of using these methods should be familiar. The constructor, of course, has been called implicitly when the object *`flossy`* was declared.

The implementation file

The implementation file contains the definition of all the methods of a class – what each one of these amounts to is, in fact, a very small C++ program. The file will almost certainly be required to have the same extension as C++ programs on your system, so you'll need to quickly check your *Local guide* for confirmation and details.

As with any C++ program, the implementation file starts with a header block of comments describing the purpose of the class and giving the name of the author, the date of writing and the platform. Any system libraries that are used in any of the methods are also included at the top, as is the header file containing the definition of the class to be implemented. Finally, the line:

```
using namespace std;
```

as usual tells the compiler that the methods will use the standard C++ names and functions.

You should by now have a template file for your C++ programs, and it is probably a sensible idea to create a copy and adapt it slightly as a template for your implementation files. It might look like this:

```
// class.cc
//
// Implementation of the < ... > class.
//
// Class to ...
//
// Author: AMJ
// Date:
// Platform:

#include <iostream>

#include "class.h"

using namespace std;
```

As a completed example, an implementation file containing the definition of a class to contain the methods for the *Sheep* class would start:

```
// sheep.cc
//
// Implementation of the Sheep class.
//
// Demonstrates the basic structure of a C++
//   implementation file.
//
// Author:   AMJ
// Date:     27th November 2001
// Platform: g++, Linux

#include <iostream>
#include <string> // Sheep's name is a string

#include "sheep.h"

using namespace std;
```

The remainder of the file contains the definitions of the methods, one at a time. It is usual for them to be implemented in the same order as they appear in the header file (it makes them easier to find when changes need to be made or problems need to be fixed), so the constructor is normally written first.

The constructor

This is the method responsible for creating an instance of the class. It is called implicitly when a class instance is declared, so the declaration:

```
Sheep flossy;
```

implicitly calls the constructor for the *Sheep* class. Happily almost all the tasks that need to be carried out by the constructor are carried out just as implicitly, and very often the programmer needs to add no lines at all to complete the constructor.

The constructor is a method with the same name as the class; the constructor for the *Sheep* class is called *Sheep*, for example. It appears in the header file simply as:

```
Sheep ();
```

To show that methods belong to a particular class, they also have the name of the class and two colons added before them in the implementation file. This gives the slightly odd looking full name for the constructor for the *Sheep* class of *Sheep::Sheep*. This forms the first part of the header of the definition of the method:

```
Sheep::Sheep
```

The name of the method is followed immediately by the parameters, enclosed in the usual brackets and separated with commas. Constructors can have parameters, but if they do not then the brackets are empty:

```
Sheep::Sheep ()
```

This line looks in fact very like the line defining the method in the header file, but it is important to remember that it doesn't have a semi-colon at the end.

The following lines form the body of the method. They resemble a C++ main program and are also surrounded by braces:

```
Sheep::Sheep ()
{
}
```

Some programmers might prefer to include a single line, perhaps to emphasise that the constructor is intentionally empty:

```
Sheep::Sheep ()
{
  return;
}
```

This is actually all that the constructor needs to do! Of course, a lot will be happening behind the scenes as memory will be being allocated for all the private values, but all this is being taken care of automatically and implicitly.

Sometimes a programmer will want to add some program statements into the constructor; the most common case is when some initial values need to be supplied for some or all of the private values. In this case it would probably be reasonable to set the initial position of a newly declared *Sheep*:

```
Sheep::Sheep ()
{
  x = 0;
  y = 0;
}
```

or:

```
Sheep::Sheep ()
{
  x = 0;
  y = 0;
  return;
}
```

Since it is part of the class the constructor is allowed to access the private attributes directly, as shown. There's more on the `return` statement in a moment.

That's all for constructors. Selectors are not much more complicated.

Selectors

The selectors can also access the private parts of the class. A selector's job is to provide the value of a given attribute to the program that calls the method. As with a constructor, the declaration starts with the name of the selector method joined to the name of the class with two colons. Selectors do not have parameters, so there is also an empty set of brackets. The return type, which is the type of the attribute that is being returned, is added to the start of the line. For example, the *getName* method of the *Sheep* class returns the name of the sheep, a string:

```
string Sheep::getName ()
```

Braces once again surround the body of the method. There is usually only one line in the body of a selector, a line that simply returns the value:

```
string Sheep::getName ()
{
  return name;
}
```

The `return` statement has appeared before (it's been at the end of every program), but this particular use is new. As the name suggests, its purpose is to terminate the execution of the method (returning to the place from where the method was called) by returning the value that was requested.

A selector will never change the value of any of the private attributes; all the method does is find and return the value. It is therefore a good idea to make this explicit by adding a `const` to the end of the method header to show that calling the method will make no changes. The full method is then:

```
string Sheep::getName () const
{
  return name;
}
```

The first header line of the definition of the method must match the corresponding line in the header file exactly[1]. In this case the header file would contain:

```
string getName () const;
```

Mutators

Mutators are very similar to selectors. The main difference is that they each have the new value for the private attribute provided as a parameter. This parameter is listed together with its type (and is given an identifier) in the brackets after the method's name. The heading line is then complete:

```
void Sheep::setName (string newName)
```

The name of the method and the types of any parameters must once again match *exactly* with the definition of the method in the header file. If there is the smallest difference the compiler will not realise that this implementation and definition are for the same method and will generate many error messages. The one thing that does not need to match is the identifier of the parameter; that can be anything at all (but it should be meaningful).

The declaration of the parameter in the brackets looks rather like the declaration of a variable and this is in fact just what it is. The parameter is available inside the method in just the same way as a variable.

The body of the mutator method takes the value of the parameter and assigns it to the private attribute. Finally, the method must return a value; mutators return `void`, so the `return` statement is used on its own[2]. The full method is now:

```
void Sheep:setName (string newName)
{
```

---

1 A tip is to create the implementation file from a copy of the header file. This can save a lot of typing and can help to make sure that all the definitions match up correctly.

2 It is possible to omit this `return` and just to let the method end. It's another style issue, and my style is to include it (as I would have done in the constructors earlier on). As with all the style issues we've seen so far, it's your choice whether to include it or not; the important thing, as always, is that you're consistent about whether you include it.

```
    name = newName;
    return;
}
```

More selectors

Selectors do not necessarily have to return a value that is actually stored in the private part of the class. It can sometimes be useful and very powerful to implement selectors that do not do this; selectors can return a value that is based on the values in the class. You might remember that we suspected that some of the *getValue* methods in the previous chapter were doing just this.

The version of the *Sheep* class in this chapter has a selector method that returns the value of the sheep, and the value is stored as an attribute. This appears in the header file as:

```
float getValue () const;
```

and would be implemented as:

```
float Sheep::getValue () const
{
  return value;
}
```

Suppose that a user of this class often wanted to use the prices of the sheep after some tax (calculated as some known percentage of the actual value) had been added. One approach to this would be to add a new private attribute to store the price plus tax and to add a new selector and mutator. There is a drawback with this: the price and the price after tax are the same information stored in different ways, so the mutator that changed one would have to change the other. If the price were reduced then the price after tax would have to be reduced too. This is a needlessly complex solution and one that would be best avoided.

Far better would be to simply create a new selector to return the price after tax. This can be added to the header file:

```
float getValuePlusTax () const;
```

and, assuming a 10% tax rate, it can be implemented:

```
float getValuePlusTax () const
{
  return value * 1.1;
}
```

A programmer using the class does not need to know whether the price with tax is stored as a private member or not (as we did not in the previous chapter). As before, the programmer just has to understand and use the public interface to the class.

Data hiding

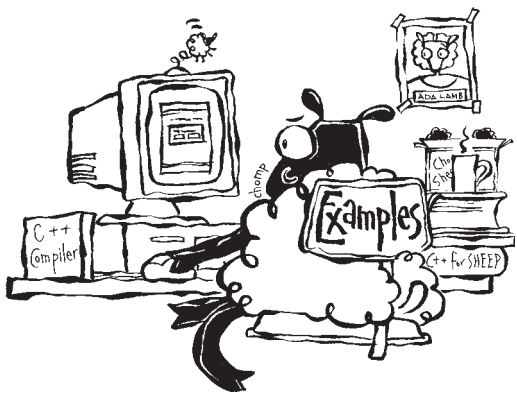
The key idea in the chapter has been *data hiding*. This is also one of the key concepts in object-oriented programming. The idea is quite simple. A programmer using a class needs to have no knowledge of how that class is actually implemented, or of how the private attributes are actually stored. In C++ terms this means that a programmer needs only know the public interface to the class and has no need to know anything at all about the private parts.

So a programmer developing a class can specify exactly how that class can be used. If the class is written and tested well it should not be possible to use it in a way that would produce incorrect results. Testing is also made easier because the interface is probably quite small; it is much easier to thoroughly test a small well-defined interface than it is to test a large unstructured program.

There is another potential benefit. If a programmer using a class needs to know nothing of the way in which the class itself is stored then this storage can be changed if needed. It can be changed, and programs using the class will still work and will be totally unaware of the change as long as the methods making up the public interface continue to behave as before.

Data hiding is achieved in C++ with selectors and mutators. These control access to the private part of the class and mean that the attributes in these private parts can be changed without the change causing any problems to programs using the class.

Selectors and mutators, together with the constructor, are the most fundamental methods to be implemented for any class. It's time for some practice.



Example 1 – More from Mollie's library

Mollie's first efforts to produce a program to store details of the borrowers from her library produced this header file:

```
#include <string>

using namespace std;

class Borrower {
  private:
    string name;  // Name of borrower
    int number;   // Unique borrower number

  public:
    Borrower ();

    void print () const;
};
```

She now wants to extend this header file to include suitable selectors and mutators. The next step is to implement the class. What should the complete header file be? What is the implementation file?

The class definition needs two selectors and two mutators, one of each for each of the private attributes. This also seems to be a good time to add some comments into the file. It becomes:

```
// borrower.h
//
// Definition of a book borrower class.
//
// Chapter 12, Examples 1.
//
// Author:   AMJ
// Date:     27th November 2001
// Platform: g++, Linux

#include <string>

using namespace std;

class Borrower {
  private:
    string name;  // Name of borrower
    int number;   // Unique borrower number

  public:
    Borrower ();

    // Selectors
    string getName () const;
    int getNumber () const;

    // Mutators
    void setName (string);
    void setNumber (int);

    // Other Methods
    void print () const;
};
```

There seems to be no particular need for the constructor to set any default values, and the selectors and mutators follow the usual patterns from this chapter.

The `print` method is neither a selector nor a mutator, but it is now possible to implement it. The header is built up as for the methods in this chapter:

```
void Borrower::print () const
```

and the body will contain nothing more than some suitable `cout` statements to display the borrower's details in some neat format. For example, to print the borrower's name followed by their number in brackets:

```
void Borrower::print () const
{
  cout << name << '(' << number << ')';
  return;
}
```

The full implementation file would be as follows. As this is a final version of the file some comments have been added to each method make sure that the method's purpose is absolutely clear.

```
// borrower.cc
//
// Implementation of a book borrower class.
//
// Chapter 12, Examples 1.
//
// Author:   AMJ
// Date:     27th November 2001
// Platform: g++, Linux

#include <iostream>
#include <string>

#include "borrower.h"

using namespace std;

// Constructor
Borrower::Borrower ()
{
}

// Selector for Name
string Borrower::getName () const
{
  return name;
}

// Selector for Number
int Borrower::getNumber () const
{
  return number;
}

// Mutator for Name
void Borrower::setName (string newName)
{
  name = newName;
  return;
}

// Mutator for Number
void Borrower::setNumber (int newNumber)
{
  number = newNumber;
  return;
}

// Method to print name followed by number in brackets
void Borrower::print () const
{
  cout << name << '(' << number << ')';
  return;
}
```

Example 2 – More wandering sheep

The whereabouts of his sheep in his field continues to be of great concern to Farmer Strawstrap. He has a program that uses a `Sheep` *class with the definition:*

```
#include <string>

using namespace std;

class Sheep {
  private:
    string name;
    int x, y; // Position in the field

  public:
    Sheep (); // All sheep start at 0,0

    // Selectors
    string getName () const;
    int getX () const;
    int getY () const;

    // Mutators
    void setName (string);
    void setX (int);
    void setY (int);

    // Other Methods
    void moveTo (int, int); // Move sheep to x, y

};
```

Foolishly the farmer has lost the implementation of the class. The implementation must be rewritten!

Taking each method in turn, the first is the constructor. The comment in the header file requires that each sheep is given the initial position (0, 0), but there seems to be no need to set any initial values for any of the other attributes. The constructor is simply:

```
Sheep::Sheep ()
{
  x = 0;
  y = 0;
}
```

The selectors and mutators all follow what is by now a very familiar pattern. An example of each:

```
// Selector for a sheep's x value
int Sheep::getX () const
{
  return x;
}

// Mutator for a sheep's name
void Sheep::setName (string newName)
{
  name = newName;
  return;
}
```

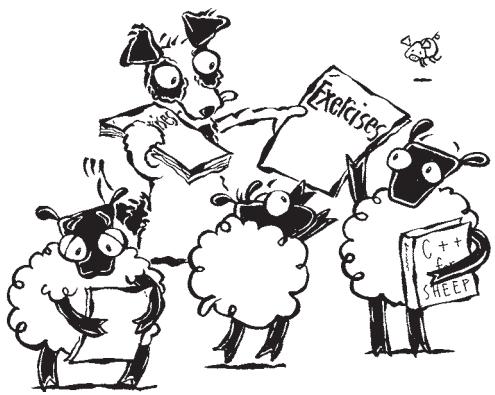
The final method is slightly more interesting:

```
void moveTo (int, int); // Move sheep to x, y
```

This method moves a sheep to a new position so it is effectively nothing more than a mutator that operates on two private attributes at the same time. It is implemented:

```
void Sheep::moveTo (int newX, int newY)
{
  x = newY;
  y = newY;
  return;
}
```

A side issue here is that this implementation of the class would work with any existing programs that used the implementation that Farmer Stroustrup has lost. The public interface hasn't been changed, and it doesn't matter whether the implementation is identical to the lost one just as long as it works correctly.



12.1 A class written to process the details of pigs has the following definition:

```
class Pig {
  private:
    string name;
    int age;
    float value;  // in UK Pounds

  public:
    Pig ();       // sets no default values

    string getName () const;
    int getAge () const;
    float getValue () const;

    void setName (string);
    void setAge (int);
    void setValue (float);
};
```

Write the implementation file for this class.

12.2 Now write a short program that will allow you to test the class. The program should prompt a user to enter some values for the attributes of an

instance of the class and should then use and test all of the methods in turn. A program such as this is called a *driver* program, and such programs are often written to test a class.

12.3 A program that has been developed to test this class contains the following lines of code. What error has been made? How should the error be corrected?

```
Pig peter;

peter.setName ("Peter");
cout << "Name has been set to " << peter.name
     << endl;
```

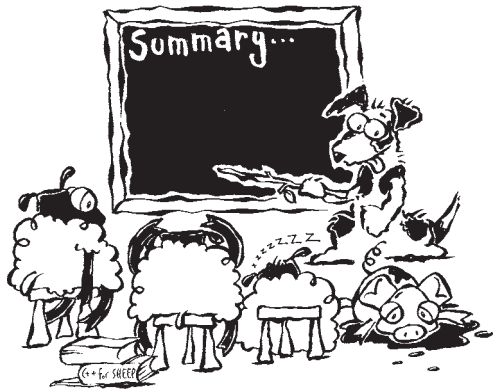
12.4 There is no error in this class definition but there is some poor design. Explain the poor design and write a corrected header file.

```
class Pig {
  private:
    string name;
    int age;
    float value;
    float valuePlusTax;   // value plus 15%
    bool hasBeenSold ();
    Pig mother;

  public:
    string getName () const;
    int getAge () const;
    float getValue () const;
    float getValuePlusTax () const;
    bool getSold () const;
    Pig getMother () const;

    void setName (string);
    void setAge (int);
    void setValue (float);
    void setValuePlusTax (float);
    void setSold (bool);
    void setMother (Pig);
};
```

12.5 In the examples you have seen so far there have been mutators and selectors for each and every private attribute. Is this always sensible? Do you think that mutators and selectors should be added to the design of every class?

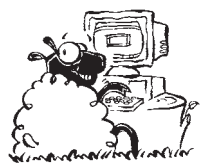


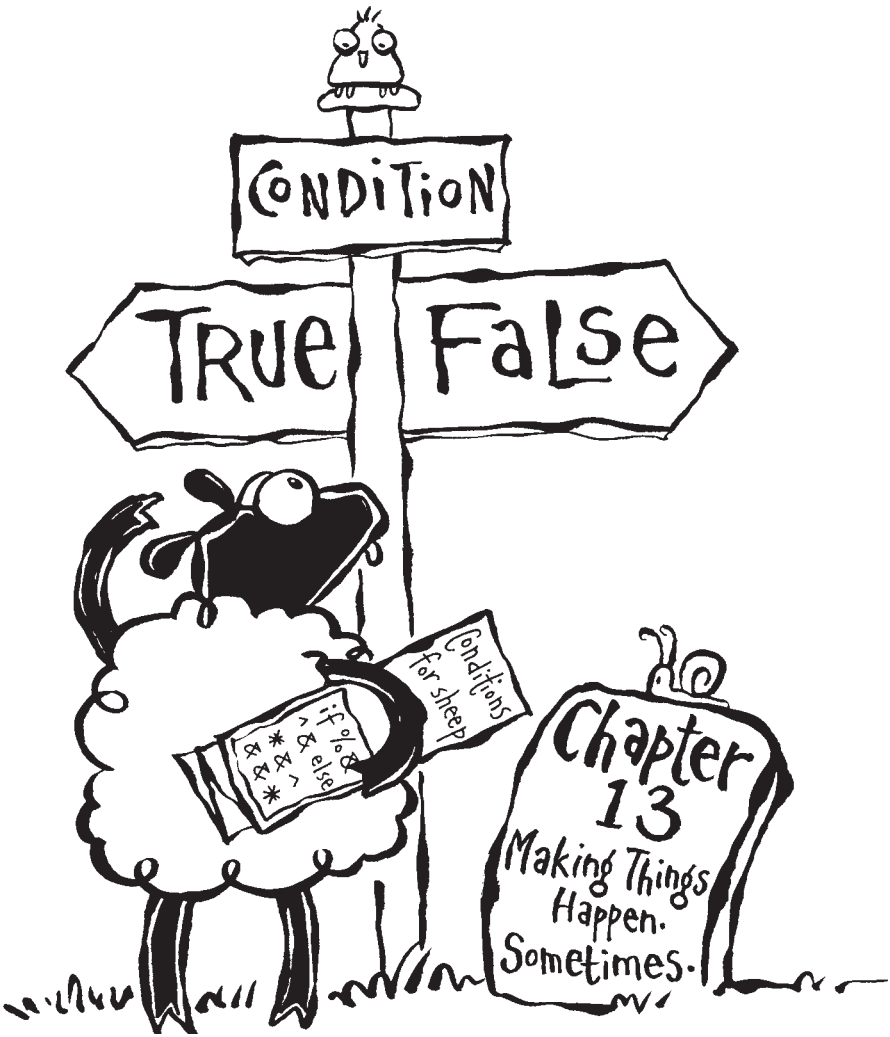
You have now seen and used both aspects of creating a C++ class. You should be able to define a class in a header file and then produce a corresponding implementation file. Finally, you should be able to write programs that make use of your class.

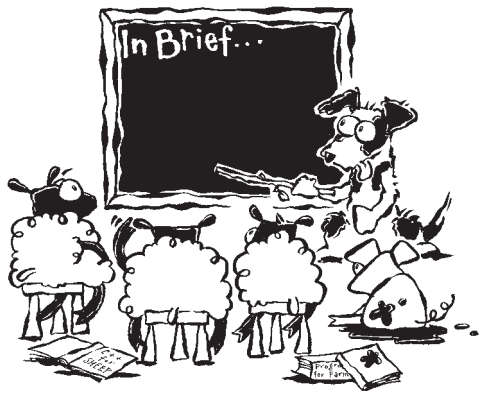
You should understand the concept of data hiding. You should know why it is important that access to the private attributes of a class is controlled and why such access should be allowed only through the class's public interface. This interface is implemented in C++ with selectors and mutators; these are simple methods that either return or change the value of a private attribute.

Although you have seen only simple examples of methods so far, the details you have seen and practised remain the same as methods become more complicated. You have in fact seen all the C++ that is used to define, implement and use very simple classes.

The remainder of this book will show you C++ that will allow you to write more and more complicated and useful programs.



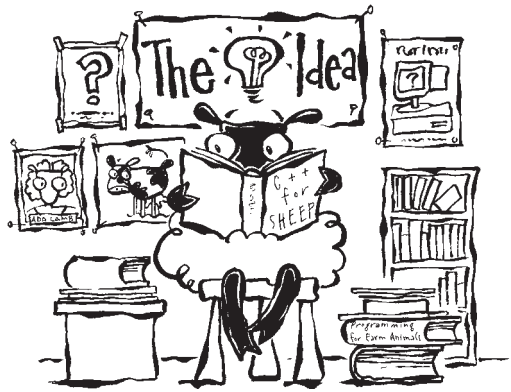

CONDITION
TRUE
FALSE
Conditions for sheep
if
else
Chapter 13
Making Things Happen. Sometimes.



Up to this point the behaviour of your programs has been very predictable. The programs have been simply a sequence of statements that are executed in a strict order from the top of the program to the bottom. While you have been able to accept input values from your users, all you have been able to do is use these values in calculations. You have not been able to use them as the basis of decisions. More complex programs take values from the user and will behave differently depending on the values that are entered. This chapter is about writing programs that do just that; they take values, test *conditions*, and then behave differently depending on the results.

After reading this chapter you should understand how to include conditional statements in your programs. You should understand the concept of a Boolean value more fully and you should be able to use Boolean variables and expressions in your programs. You should be able to combine these values using Boolean operators and you should be able to use these values and operators in conditional statements. You should understand and be able to use the C++ `if` and `switch` statements.

The examples in this chapter all illustrate how to use conditional statements in complete C++ programs. Conditional statements can, of course, be used in methods or classes – you will be asked to try this out in some of the exercises.



A *condition* is a statement that is either true or false. Programs need to be able to test conditions so that they can behave differently in different situations.

This is achieved with *conditional statements*; these are statements that test a condition and then choose between two (or more, in fact) different possible behaviours depending on whether the condition was found to be true or false. Conditional statements can be combined to allow programs to make a choice between more than two possible behaviours.

There are two types of conditional statement in C++. The most common is the `if` statement; there is also the `switch` statement, which provides a shorthand form for a particular type of `if` statement. It is possible to write all `switch` statements as `if` statements, but the `switch` statement can be more elegant and neater. Before using either of these statements, however, you need to understand the concepts of true and false and how these can be stored and expressed in Boolean variables.

True and false

Some statements are always either true or false. Here are some examples:

Statements that are true

- Flossy is a sheep.
- Potatoes grow in the ground.
- Christmas Day is the 25th of December.
- Sheep live in fields and eat grass.

Statements that are false

- Flossy is a llama.
- Potatoes grow on trees.
- Christmas Day is the 14th of June.
- Sheep live in trees and eat pizza.

All of the statements are either true or otherwise they are false. They must be one or the other. Their "true-ness" will never change no matter what happens.

Some statements are more complicated. Some statements are true some of the time and false at other times. They are always one or the other and never, of course, both. Here are some examples:

- Flossy is eating some grass in her field.
- I have some potatoes in my kitchen.
- It is Christmas Day tomorrow.
- There are six sheep in the field and they are all eating grass.

At any point in time these statements will either be true or false. Their "true-ness" depends on some condition:

- Flossy may be eating grass in her field, or she may not be.
- I may have some potatoes in my kitchen, or I may have run out.
- It may be Christmas Day tomorrow, or it might be the middle of June.
- There may be six sheep in the field, but some more may arrive, or some may go away. The sheep that are there may be eating grass, or some or all of them may stop.

In order to determine the "true-ness" of the statement it is necessary in each case to test this condition. This test could be expressed in pseudo-code, for example:

```
IF IT IS DECEMBER 24TH TODAY THEN
  IT IS CHRISTMAS DAY TOMORROW
OTHERWISE
  IT IS NOT CHRISTMAS DAY TOMORROW
END IF
```

Or perhaps:

```
IF IT IS DECEMBER 24TH TODAY THEN
  IT IS TRUE THAT IT IS CHRISTMAS DAY TOMORROW
OTHERWISE
  IT IS FALSE THAT IT IS CHRISTMAS DAY TOMORROW
END IF
```

Or in more general terms:

```
IF THE CONDITION IS TRUE THEN
  THE STATEMENT IS TRUE
OTHERWISE
  THE STATEMENT IS FALSE
END IF
```

This is totally unambiguous; the condition is either true or it is false and so, therefore, is the statement.

Boolean variables and operators

In C++, Boolean variables are variables that can store one of two values: `true` or `false`. The data type is `bool` and they are declared in the same way as all other variables:

```
bool finished;
```

Boolean variables can only be assigned one of the two possible values:

```
finished = true;
finished = false;
```

On their own Boolean variables are of very limited use, which is why we have not made any use of them so far. It is not possible to read values for them from the user's keyboard and it is not possible to output their values directly[1]. Their power comes when they are used in *Boolean expressions*.

Variables of the more familiar numeric types have a number of operators that can be used with them. You have seen (and hopefully written) many programs where integers and floating-point values have been added together, subtracted, multiplied or divided. Boolean values also have a set of operators; their meaning is similar to the operators for the numeric types, but the meaning is now set in terms of logical operations. These operators allow Boolean variables and values to be combined in various ways based on the principles of mathematical logic. You may well have met them before in a mathematics course[2].

There are three basic Boolean operators:

---

1 This is another one of those things that is not strictly true. Some versions of C++ will allow you to output a Boolean value using `cout << boolalpha <<` followed by the identifier of the Boolean.

2 You might also have met them in a computer hardware or electronics course. Combining Boolean operations is effectively the same as combining currents with "gates" in a circuit.

and
The "and" operator combines two Boolean values to give the result `true` if and only if the two values are both `true`. Boolean expressions are often written out in *truth tables* – here is the truth table for the "and" operator.

First	Second	First *and* Second
`false`	`false`	`false`
`false`	`true`	`false`
`true`	`false`	`false`
`true`	`true`	`true`

or
The "or" operator combines two Boolean values to give the result `true` if and only if either (or indeed both) of the two values is `true`:

First	Second	First *or* Second
`false`	`false`	`false`
`false`	`true`	`true`
`true`	`false`	`true`
`true`	`true`	`true`

not
The "not" operator takes just one Boolean value and reverses it. It gives the result `true` if the original value is `false` and vice versa:

Original	*not* Original
`false`	`true`
`true`	`false`

These three operators have the following symbols in C++[3]:

and	`&&`
or	`||`
not	`!`

The symbol used in the "or" is the "pipe" symbol, which you may not have met before. This is normally found on the keyboard to the left of the *z* key or else just above the *TAB*. It may well be shown on your keyboard as two vertical lines with a gap.

The symbol for "not" is the exclamation mark. For reasons that are not especially clear, in computing circles it is normally pronounced, "bang".

3 Recent versions of C++ actually allow the use of `and`, `or` and `not` as words. It is still far more common to use the symbols, which is what all the programs here will do.

The two operators that take two values ("and" and "or") are used in much the same way as the similar mathematical operators for the numeric data types. The "not" operator is simply added before the identifier of the variable:

```
bool conditionOne = true;
bool conditionTwo = false;

// (conditionOne && conditionTwo) is false
// (condition One || conditionTwo) is true
// !conditionOne is false, !conditionTwo is true
```

Boolean values expressed in this way are the key to using conditional statements.

Comparison operators

The key to using conditional statements is that they give the ability to compare two values. A given value must be compared to some expected value to build an expression which can then be used to determine whether a condition is true or false. This comparison requires some operators.

The following operators are available in C++. The symbols and meanings of most of them are much the same as you are probably familiar with from mathematics.

==	equal to
!=	not equal to
>	greater than
<	less than
>=	greater than or equal to
<=	less than or equal to

A common error is to confuse the "equal to" comparison operator == with the assignment operator =. You have been warned! Remember that a good way of remembering the difference is to get into the habit of reading the assignment operator as "becomes" or "is assigned" rather than the potentially misleading "equals". For the same reason you might want to read the comparison operator as "is the same as".

The meaning of each of these operators should be fairly obvious, but they are illustrated in this sample code:

```
int first, second;

first = 10;   // first is assigned 10
second = 20;  // second is assigned 20

// first == 10      this is true, first equals 10
// first == second  this is false, first does not equal second
// second != 10     this is true, second does not equal 10
// first > 0        this is true, first is greater than 0
// second < 0       this is false, second is not less than 0
// second < 20      this is also false, second is not less than 20
// second <= 20     but this is true, second is equal to 20
// first >= 10      this is true, first is equal to 10
```

Each one of these comparisons is itself actually a Boolean value (or, more correctly, a Boolean expression); each one is either true or false. This means that it is possible to assign such a comparison to a Boolean variable. For example:

```
int first = 10;
bool firstIsTen;

firstIsTen = (first == 10); // assigns true
```

The brackets in this assignment are not essential, but they do make the line a lot clearer. Some more examples:

```
int aNumber = 10;
bool condition;

condition = (aNumber > 0);      // assigns true
condition = (aNumber <= 100);   // assigns true
condition = (aNumber != 10);    // assigns false
condition = (aNumber == 0);     // assigns false
```

Conditions, comparisons and Boolean variables are only really useful if they can be used in conditional statements. So it's time to look at the first of these.

The `if` statement

In its simplest form, the `if` statement evaluates a condition and executes the statements following it only if the condition is found to be true. An `if` statement starts with the word `if`, which is then followed by the condition in brackets:

if (<condition>)

The statements affected by the `if` statement are below, enclosed in the customary braces:

```
if (<condition>) {
   // statements executed only if condition is true
}
```

If there is only a single statement inside the braces they can be omitted, but it is always good practice to include them to make sure that things are clear. The statements inside the braces should also always be indented to clearly show that the conditional statement controls them.

For example, if a program were required to divide two numbers it is important to check that the divisor is not equal to 0, since dividing by 0 gives an error (an error that we have already seen in Chapter 9[4]). This can be achieved simply enough with an `if` statement:

```
float top, bottom;

cout << "Enter the first number: ";
cin >> top;
cout << "Enter the second number: ";
cin >> bottom;
```

4 Where I promised that here in Chapter 13 we would see how to avoid this error. And here we are!

```
if (bottom != 0) {
  cout << top << " divided by " << bottom
       << " is " << top / bottom << endl;
}
```

This makes sure that the calculation takes place only if it is safe to do so. The statement inside the braces is executed only if the condition:

```
bottom != 0
```

is true, so it is executed only if the divisor does not equal 0.

Of course, if the divisor was 0 it would be useful to print a message to the user telling them why no output has been produced; as matters stand the user would just be left with no response at all from the program, a very bad example of human–computer interaction. This can be done using the optional `else` part of an `if` statement.

The `else` clause of the `if` statement

The `else` part of the `if` statement specifies the statements that will be executed if the condition is found to be false. These statements are enclosed as usual by braces and are also indented:

```
if (<condition>) {
  // statements executed if condition is true
}
else {
  // statements executed if condition is false
}
```

A more complete statement to check that a division by 0 does not happen would be:

```
if (bottom != 0) {
  cout << top << " divided by " << bottom
       << " is " << top / bottom << endl;
}
else {
  cout << "Error: Cannot divide by zero!" << endl;
}
```

Since there is a choice involved here, this could also (and equally well) be written:

```
if (bottom == 0) {
  cout << "Error: Cannot divide by zero!" << endl;
}
else {
  cout << top << " divided by " << bottom
       << " is " << top / bottom << endl;
}
```

The difference in the second example is that the condition that is tested has been "reversed".

The condition in the brackets of an `if` statement will often combine two connected tests. Suppose that a program requires that the user enter a value within a certain range such as between 0 and 20; the program must test the value and reject it if it is less than 0 or if it is greater than 20.

The description of the problem shows that this will be an application for the Boolean "or" operator. Assuming that the range is inclusive the code is:

```
int aNumber;

cout << "Enter a number between 0 and 20: ";
cin >> aNumber;

if (aNumber < 0 || aNumber > 20) {
  cout << "Error: Invalid Value!" << endl;
}
else {
  // Process valid value
}
```

A common error is to write a statement such as this in this way:

```
if (aNumber < 0 || > 20) { // THIS IS AN ERROR!!
```

While this may make sense, especially when read aloud, it is always an error. The reason should be obvious. The value to the right of the "or" operator is "> 20"; this is not a properly formed Boolean condition since there is nothing for the 20 to be compared to. It should be obvious that this is an error:

```
if (> 20) { // THIS IS ALSO AN ERROR!!
```

and this is the exact condition that is on the right-hand side of the "or" operator in this example.

While validating this input, much the same effect could be achieved by accepting only those values that are greater than or equal to 0 and less than or equal to 20. Once again the description contains the name of the required Boolean operator, in this case "and":

```
if (aNumber >= 0 && aNumber <= 20) {
  // Process valid value
}
else {
  cout << "Error: Invalid Value!" << endl;
}
```

This example is exactly equivalent to this:

```
if (aNumber > -1 && aNumber < 21) {
  // Process valid value
}
else {
  cout << "Error: Invalid Value!" << endl;
}
```

The choice of which of these possibilities to use is largely a matter of the programmer's taste and personal programming style. With experience, in most cases one possibility will feel more natural, but this does not mean that any of the others are ever wrong.

More complex `if` statements

Sometimes the choice to be made is more complex. There may be three possible values for some variable and three possible paths for the program to follow. There may be many more. The `if` statement is easily extended to deal with this:

```
if (<condition1>) {
  // statements if condition1 is true
}
else if (<condition2>) {
  // statements if condition1 is false but condition2 is true
}
else {
  // statements if condition1 and condition2 are both false
}
```

Only one of the three possibilities is ever executed. If the first condition is true the first set of statements is used. The second condition is only tested if the first condition is found to be false; the statements that it controls are executed only if the second condition is found to be true. Finally, if both conditions are false, the final set of statements will be used. One of the three sets of statements is therefore always executed.

As an example of using a more complex `if` statement, consider the problem of printing a date. Suppose the date is stored as three integers, one each for the day, the month and the year, and that the required format prints the day part as "1st", "2nd", "3rd" and so on. The code to generate the required two letters to follow the number would be a single long `if` statement. Assuming that the day is held in the integer variable *day*, one possibility to output the day part would be:

```
cout << day;

if (day == 1 || day == 21 || day == 31) {
  cout << "st";
}
else if (day == 2 || day == 22) {
  cout << "nd";
}
else if (day == 3 || day == 23) {
  cout << "rd";
}
else {
  cout << "th";
}
```

There can be any number of conditions tested in this way. There does not need to be a final `else` section and there need not be if the tests beforehand cover every possibility. If there is no `else` section it is then possible that none of the possibilities will ever be executed.

This form of the `if` statement is so common that there is a different conditional statement available; the `switch` statement provides a neat mechanism for particular types of comparisons. That said, the `switch` statement does nothing that cannot be expressed with a suitable `if` statement.

The `switch` statement

The `switch` statement provides no control that an `if` statement does not; it simply allows for neater code. It is used in the particular case where there is a set of possible values for some variable (called the *control variable*) and different actions must be taken for each one. The format is:

```
switch (<control variable>) {
  case <value 1>:   // actions if variable is value 1
                    break;
  case <value 2>:   // actions if variable is value 2
                    break;
  default:          // actions if variable had none of the values listed
}
```

Since it must be possible to list all the possible values of the control variable, if follows that the variable must be a char or an int[5]. It is not possible to list all possible strings or all possible floating-point numbers. The effect of the statement is that the case containing the value is found, and the statements found there are executed. The end of the case is marked with `break`, and execution stops.

A `switch` statement would be suitable for printing the name of the month from a numeric representation of a date. There would be twelve possibilities, one for each month. Assuming that the month were stored in an integer variable *`month`* the code would be:

```
switch (month) {
  case 1: cout << "January";
          break;
  case 2: cout << "February";
          break;
  case 3: cout << "March";
          break;
  case 4: cout << "April";
          break;
  // and so on for the other months
}
```

There would be no need for a `default` section in this case because all the possible values would presumably be known and listed. If it were possible that an invalid value could be held in *`month`* there are two possibilities. One is to add a `default` section:

```
default: cout << "Error: Invalid Month!";
```

A potentially neater alternative is to keep the `switch` statement as it stands and to use it only if the value is found to be valid:

```
if (month >= 1 && month <= 12) {
  switch (month) {
    // switch cases
  }
}
```

Only variables whose possible values can be listed can be used in `switch` statements; this effectively means only integers and characters. The `default` case means that it is not necessary to list all the possible values, of course.

As a further example, here is a `switch` statement that would expand the names of the points of the compass held in the `char` variable *`point`*:

---

5 Formally, these are called ordinal types since their values can be listed and ordered. Booleans are ordinal too, but since there are only two possible values there is little point in using a `bool` in a `switch` statement.

```
switch (point) {
  case 'N': cout << "North";
            break;
  case 'S': cout << "South";
            break;
  case 'E': cout << "East";
            break;
  case 'W': cout << "West";
            break;
}
```

Sometimes the same statement is required for more than one value. If in this compass example the character could be in upper or lower case then the same statement should be executed for 'n' as for 'N', and so on. It would be wasteful to duplicate the code, so more than one value can be attached to a case:

```
switch (point) {
  case 'N':
  case 'n': cout << "North";
            break;
  case 'S':
  case 's': cout << "South";
            break;
  case 'E':
  case 'e': cout << "East";
            break;
  case 'W':
  case 'w': cout << "West";
            break;
}
```

If a `switch` statement like this is used, it is important to make sure that the `break` lines are used correctly to separate the various cases. It is also clear why omitting a `break` in a simpler `switch` statement can produce all sorts of unexpected results.

The neatness of the `switch` statement is apparent from the `if` statement that is equivalent to this last example:

```
if (point == 'N' || point == 'n') {
  cout << "North";
}
else if (point == 'S' || point == 's') {
  cout << "South";
}
else if (point == 'E' || point == 'e') {
  cout << "East";
}
else if (point == 'W' || point == 'w') {
  cout << "West";
}
```

At the same time the `switch` statement does nothing that an `if` cannot. Once again the choice of whether or not to use a `switch` statement is often just a matter of taste or preferred programming style.

Ending a program

Conditional statements will allow you to check that the values supplied by the users of your programs are as expected. This raises the question of what

to do if the values are unexpected; often the error will be fatal and there will be little point in carrying on asking for the rest of the values and making calculations[6].

If a program requires a single value it is simple to just exit the program if an invalid value is provided – this is precisely what the example to trap a potential division by 0 did. When more than one value is required it is possible to use the same technique to achieve this effect with a sequence of conditional statements:

```
// get value 1
if (<value 1 is not valid>) {
   // display an error
}
else {
   // get value 2
   if (<value 2 is not valid>) {
      // display an error
   }
   else {
   // process the values
   ...
   }
}
```

This will work, but will quickly become very cumbersome and unwieldy, as more and more conditional statements have to be added for more values. Imagine what it would look like if there were 10 input values, or 100!

An alternative is to use a quick and immediate way to end a program. The `exit` command provides just that:

```
exit (1);
```

This terminates the program immediately without processing any more statements. The parameter in the brackets can be used to indicate why the program has exited abruptly; this value can sometimes be trapped by the system that called the program. It is similar to the value provided by the usual `return` statement at the end of each program.

`exit` is actually part of the C language and requires an additional include library:

```
#include <cstdlib>
```

The code for dealing with invalid input values is now much simpler:

```
// get value 1
if (<value 1 is not valid>) {
   cout << "Error: Value 1 not valid" << endl;
   exit (1);
}
```

6 We will soon see a more elegant solution to this problem where we will be able to keep asking a user for values until they enter something correct. But for the moment something rather more brutal is required.

```
// get value 2
if (<value 2 is not valid>) {
   cout << "Error: Value 2 not valid" << endl;
   exit (2);
}
```

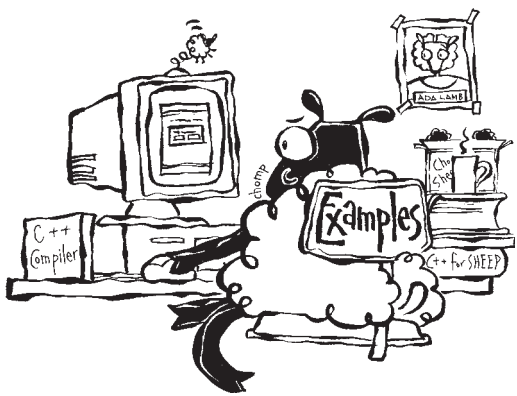
and so on. It is good practice to use different integer values for each use of `exit` to indicate the different possible reasons.

Conditional statements and conditions

Conditional statements allow a great deal of control over the way in which programs handle the values supplied by users. It is now possible to identify and handle cases of values that would previously have caused serious errors. These errors can now be dealt with neatly, with useful messages being displayed to the user.

With conditional statements it is now possible to write programs that examine the values provided by users and work in different ways determined by these values.

It's time to practise doing this with the examples and exercises.



The conditional statements described in the chapter can be used to make some of the programs from previous examples much more robust. They can be made to handle unexpected input values and to provide useful feedback to their users if errors are made.

These two examples are taken from Chapter 8.

Example 1 – Farmer Strawstrap's farm revisited

Farmer Strawstrap still wants a computer program to help him keep details of the animals on his farm. He particularly wants to know how many animals he has. He knows that on the farm at the moment there are many sheep and many pigs, but they keep moving around and he keeps losing count. He also wants to know what percentage of his animals are sheep and what percentage are pigs.

Write a program that prompts Farmer Strawstrap to enter the number of sheep and the number of pigs and then displays the total number of animals on his farm together with the percentage that are sheep and the percentage that are pigs.

Here is the solution to this example from Chapter 8:

```
// animals.cc
//
// Calculates the number of animals of the farm and
// displays the percentages that are sheep and pigs.
//
// Chapter 8, Examples 1.
//
// Author: AMJ
// Date: 20th October 2001
// Platform: g++, Linux

#include <iostream>

using namespace std;

int main ()
{
  int numSheep, numPigs, numAnimals;
  float percentSheep, percentPigs;

  // Get Input Values

  cout << "Enter the number of sheep: ";
  cin >> numSheep;

  cout << "Enter the number of pigs: ";
  cin >> numPigs;

  // Calculations

  numAnimals = numSheep + numPigs;

  percentSheep = (float (numSheep) / numAnimals) * 100;
  percentPigs = (float (numPigs) / numAnimals) * 100;

  cout << "There are " << numAnimals << " animals"
       << endl
       << percentSheep << "% are Sheep" << endl
       << percentPigs << "% are Pigs" << endl;

  return 0;

}
```

This program is fine, but it cannot cope with errors from the user. An obvious error that cropped up in the chapter on testing is that there cannot be a negative number of sheep or pigs entered. Also, while 0 is probably a valid entry for either value, it is not sensible to carry out the arithmetic if there are no animals; there is also the possibility of a division by 0 error if both values (and therefore *numAnimals*) are 0.

A conditional statement is needed to ensure that the number of sheep entered is not less than 0:

```
cout << "Enter the number of sheep: ";
cin >> numSheep;

if (numSheep < 0) {
```

```
    cout << "Error: Negative Sheep!" << endl;
  }
```

Of course this is a fatal error, so there is no need to ask for the number of pigs. Put another way, it is only sensible to carry on if this value was valid:

```
  if (numSheep < 0) {
    cout << "Error: Negative Sheep!" << endl;
  }
  else {
    cout << "Enter the number of pigs: ";
    cin >> numPigs;
    if (numPigs < 0) {
      cout << "Error: Negative Pigs!" << endl;
    }
    else {
      // do arithmetic
      // display results
    }
  }
```

This is becoming unwieldy, so it is probably neater to make use of the `exit` command. This simplifies the code somewhat:

```
  cout << "Enter the number of sheep: ";
  cin >> numSheep;

  if (numSheep < 0) {
    cout << "Error: Negative Sheep!" << endl;
    exit (1);
  }

  cout << "Enter the number of pigs: ";
  cin >> numPigs;

  if (numPigs < 0) {
    cout << "Error: Negative Pigs!" << endl;
    exit (2);
  }
```

The system calling this program might be able to distinguish between the two conditions that have caused the program to exit by examining the parameter.

A third possible condition where it would be appropriate to exit would be when both values entered are 0. This could be achieved in the same way as the two examples above, or might be handled better (and less brutally) with an `else` statement. It is again largely a matter of taste.

A complete program for this example is:

```
  // animals.cc
  //
  // Calculates the number of animals of the farm and
  //   displays the percentages that are sheep and pigs.
  //
  // Chapter 13, Examples 1.
  //
  // Author:   AMJ
  // Date:     20th October 2001
  // Platform: g++, Linux
```

```
//
// Modifications
// - Now more robust and handles entry of invalid values.

#include <cstdlib> // for exit ()
#include <iostream>

using namespace std;

int main ()
{
  int numSheep, numPigs, numAnimals;
  float percentSheep, percentPigs;

  // Get Input Values

  cout << "Enter the number of sheep: ";
  cin >> numSheep;

  if (numSheep < 0) {
    cout << "Error: Negative Sheep!" << endl;
    exit (1);
  }

  cout << "Enter the number of pigs: ";
  cin >> numPigs;

  if (numPigs < 0) {
    cout << "Error: Negative Pigs!" << endl;
    exit (2);
  }

  // Calculations

  if (numSheep != 0 || numPigs != 0) {

    numAnimals = numSheep + numPigs;

    percentSheep = (float (numSheep) / numAnimals) * 100;
    percentPigs = (float (numPigs) / numAnimals) * 100;

    cout << "There are " << numAnimals << " animals"
         << endl
         << percentSheep << "% are Sheep" << endl
         << percentPigs << "% are Pigs" << endl;
  }
  else {
    cout << "There are no animals" << endl;
  }

  return 0;

}
```

Example 2 – Flossy's pocket money revisited

Flossy still wants to use a computer program to keep track of her pocket money. She wants a program that will add the amount she receives to the amount she has left and then subtract the amount she has spent.

Additionally, Flossy would also like to know whether she has more money, less or the same left at the end of the week.

The additional requirement has not been implemented before. Here is the program from Chapter 8 that met most of Flossy's needs:

```
// money.cc
//
// Flossy's Pocket Money Program
// Calculates the amount remaining after expenditure.
//
// Chapter 8, Examples 2.
//
// Author:   AMJ
// Date:     20th October 2001
// Platform: g++, Linux

#include <iostream>

using namespace std;

const float WeeklyPocketMoney = 15.00;

int main ()
{
  float pocketMoney;
  float amountSpent;

  // Input

  cout << "How much is left from last week? ";
  cin >> pocketMoney;

  cout << "How much have you spent? ";
  cin >> amountSpent;

  // Calculations

  pocketMoney += WeeklyPocketMoney;
  pocketMoney -= amountSpent;

  // Output

  cout << "You now have " << pocketMoney
       << " left." << endl;

  return 0;

}
```

Once again it is possible to use conditional statements to make sure that the numbers entered into the program are valid. It seems reasonable to assume that the two values entered must both be greater than or equal to 0. The code to validate this is much the same as in the previous example.

For example:

```
cout << "How much have you spent? ";
cin >> amountSpent;
```

```
if (amountSpent < 0) {
  cout << "Error: Invalid Value!" << endl;
  exit (1);
}
```

Adding Flossy's new requirement is slightly more complicated. It is now necessary to compare the amount of money that is originally entered with the amount that is left after the calculations; at present the original value is lost when these calculations have been carried out.

A new variable is needed to store the original amount. This can then easily be compared to the final amount; this new variable might be called *originalMoney*. There are three possibilities to deal with: Flossy has more money, less money or the same amount. The conditional statement will need to handle all three, so some `else` clauses will be needed:

```
if (pocketMoney > originalMoney) {
  cout << "There is more left at the end of the week"
       << endl;
}
else if (pocketMoney < originalMoney) {
  cout << "There is less left at the end of the week"
       << endl;
}
else if (pocketMoney == originalMoney) {
  cout << "There is the same left at the end of the week"
       << endl;
}
```

As it happens, the condition in the final test is not really needed. If the first two conditions are both false then this third *must* be true. It is good practice to leave it in, though, to make it clear what the third condition is actually testing.

A complete program for Flossy's pocket money needs is below. Since this is a modification of the previous program, the header block has been completed to show what changes have been made.

```
// money.cc
//
// Flossy's Pocket Money Program
// Calculates the amount remaining after expenditure.
//
// Chapter 13, Examples 2.
//
// Author:   AMJ
// Date:     20th October 2001
// Platform: g++, Linux
//
// Modifications
// - Now validates both input values  (AMJ 22/10/2001)
// - Displays whether more, less or the same is left at
//   the end of the week              (AMJ 22/10/2001)

#include <cstdlib> // for exit ()
#include <iostream>

using namespace std;

const float WeeklyPocketMoney = 15.00;
```

```
int main ()
{

  float pocketMoney;
  float amountSpent;
  float originalMoney;

  // Input

  cout << "How much is left from last week? ";
  cin >> pocketMoney;

  if (pocketMoney < 0.0) {
    cout << "Error: Negative Money Left!" << endl;
    exit (1);
  }

  cout << "How much have you spent? ";
  cin >> amountSpent;

  if (amountSpent < 0.0) {
    cout << "Error: Negative Amount Spent!" << endl;
    exit (2);
  }

  // Calculations

  originalMoney = pocketMoney;

  pocketMoney += WeeklyPocketMoney;
  pocketMoney -= amountSpent;

  // Output

  cout << "You now have " << pocketMoney
       << " left." << endl;

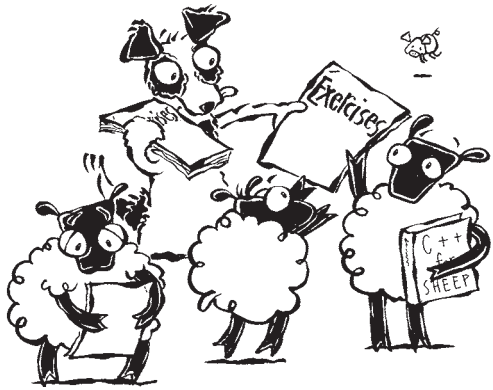
  if (pocketMoney > originalMoney) {
    cout << "There is more left at the end of the week"
         << endl;
  }

  else if (pocketMoney < originalMoney) {
    cout << "There is less left at the end of the week"
         << endl;
  }

  else {
    cout << "There is the same left at the end of the week"
         << endl;
  }

return 0;

}
```



13.1 For each of the following conditions write down a value of the variable for which the condition is true. Try to write down at least two values for which the condition is false; is this always possible?

```
aNumber == 1
aNumber >= 3.5
aChar != 'A'
aNumber < 100
aString == "flossy"
```

13.2 Examine the following program fragment carefully.

```
int aNumber;
float aFloat = 0.0;
string aString = "sheep";
aNumber = 10;
aFloat += 3.0;
```

After these statements have been executed what is the value of the following conditions?

```
aNumber == 2
aNumber <= 10
aString == "flossy"
aNumber == 10 && aFloat == 4.0
aNumber == 10 || aString == "flossy"
!(aNumber == 5)
```

13.3 Write a program that prompts the user to enter two floating-point numbers and returns the result of the first divided by the second. Your program should handle the special case where the second number is 0 and should display a suitable message informing the user that division by 0 is not possible.

13.4 Write a program that prompts the user to enter the current year and the year in which they were born and displays the user's age (or, more accurately, the age that the user will be this year). The program should check that the current year is greater than 2000 and that the user's year of birth is earlier than the current year. An informative error message distinguishing between the two possible errors should be displayed if the user does not provide valid data.

13.5 A class to store dates has the private members:

```
private:
  int day, month, year;
```

A method is required that will print the date in the format of, for example:

```
14th June 1967
23rd March 1982
10th October 1066
```

The method has the prototype:

```
void printFormatted () const;
```

Implement the class and the method. Write a short program that tests your implementation. There is no need to check that the date supplied is valid.

13.6 A simple class has been written to store the position of a sheep in a field. The header file is:

```
// sheep.h
//
// Store sheep's name and position.
//
// Chapter 13, Exercises 6.
//
// Author:    AMJ
// Date:      19th December 2001
// Platform:  g++, Linux

#include <string>

using namespace std;

class Sheep {
  private:
    string name;
    int x, y;   // position

  public:
    Sheep ();   // initialises position to 0, 0

    // Selectors

    string getName () const;
    int getX () const;
    int getY () const;

    // Mutators

    void setName (string);
    void setX (int);
    void setY (int);

    void moveTo (int, int);

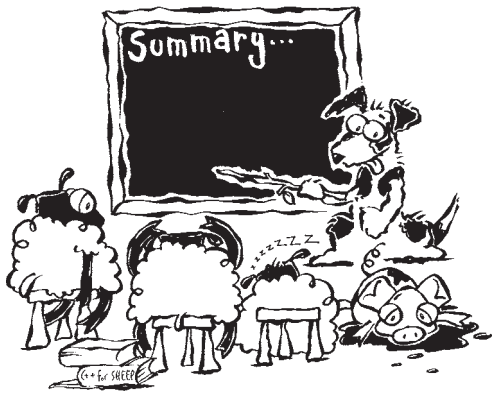
    // Methods to move the sheep
    // Parameter is number of units to move

    void moveNorth (int);
    void moveSouth (int);
    void moveWest (int);
    void moveEast (int);

};
```

Implement this class and then use it in a simple program.

The program should prompt the user to enter the name and initial location of a sheep and should then create an instance of the class. Both the x and y values in the initial location should be positive. Finally, the user should be asked to enter a direction as a single character (N, S, W or E – remember to validate it) and a number of units (which should also be positive). The program should end by displaying the position to which the sheep has been moved.



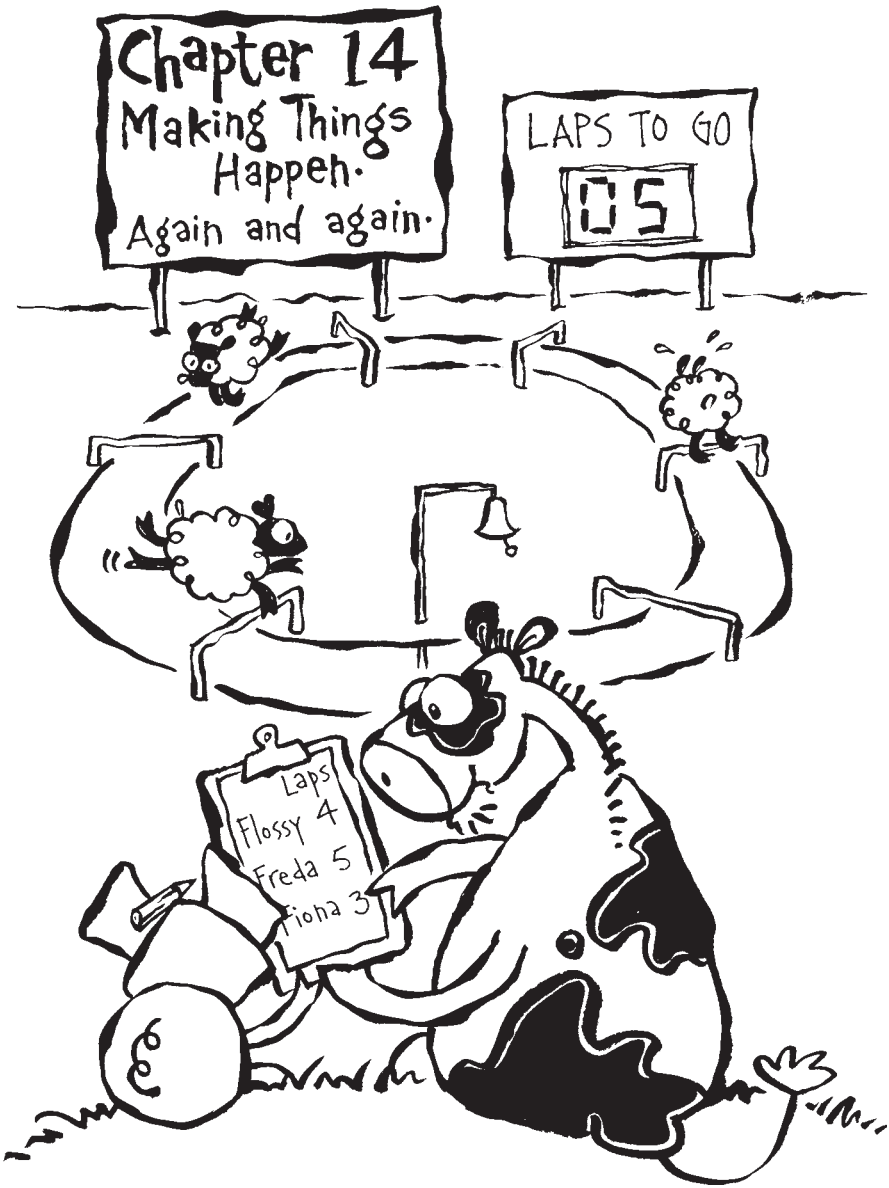
Conditional statements allow programs to react and behave differently in response to values provided by users. The key idea behind a conditional statement is a condition: an assertion that can either be true or false. A condition is always true or false, although which it is can change as a program is run.

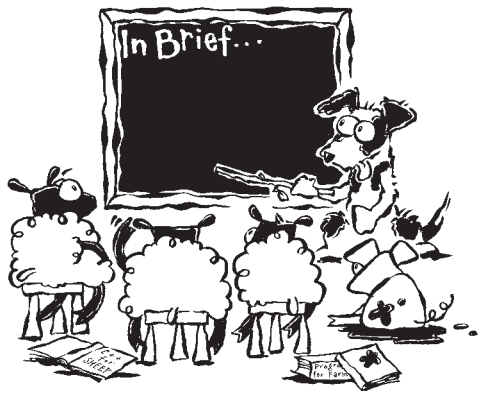
The values `true` and `false` are Boolean and can be stored in variables with the type `bool`. Boolean values and conditions can be combined using the basic Boolean operators "and", "or" and "not". Any number of conditions can be combined using these operators.

Conditions often include comparison operators. The simplest operator is "equals", represented in C++ as `==`. There is a range of other operators for comparing with values that are less than or greater than some known value.

When an unexpected value is detected in its input a program can be made to exit immediately using the `exit` command. If a program does end in this way it is always good practice to display an informative message to tell the user what has happened. A numeric value can also be passed to the computer via the `exit` command.

Of course it would be better if the program told the user of the error and gave them the chance to provide another value. A user is likely to get very frustrated at having to continually re-run a program. To allow this sort of control, statements must be executed over and over again; this leads neatly on to the topic of program loops.


Chapter 14
Making Things Happen.
Again and again.
LAPS TO GO
05
Laps
Flossy 4
Freda 5
Fiona 3

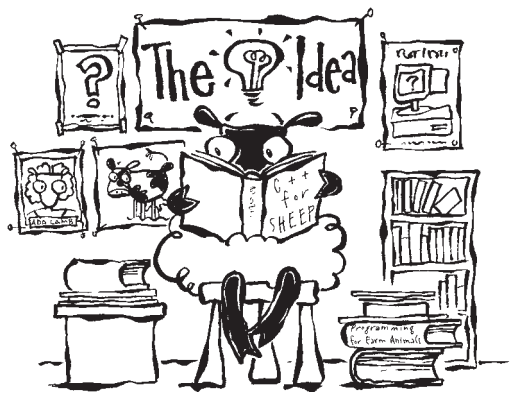


Conditional statements control the order in which the other statements in a program are executed. However, even with this level of control the fact remains that the overall flow of the program is still from the top to the bottom; there are usually several routes to the end of the program, but the flow is always in the same prevailing direction. Once a statement has been executed it can never be executed again. It is not possible to go backwards.

There is a second type of statement that can control the order of execution of a C++ program. Program *loops* allow statements to be executed a certain number of times, or while some Boolean condition is true, or until some Boolean condition has become false. This chapter explains the different kinds of loops that are available and describes how to use loops in your programs.

After reading this chapter you should be able to write programs that include loops. You should understand the different kinds of loops available in C++ and you should be able to decide which to use to solve a particular problem.

Once again, while the examples in this chapter all use complete C++ programs for convenience, loops may be used in methods. Chapter 15 will look in more detail at using more complex C++ in methods.



The previous chapter explained how to validate the values that users supply to your programs. This is effective as far as it goes, but the dialogue that has resulted is hardly very friendly. The end result is that users are simply told

that they have made an error and the program then finishes; if the user wants to try again with another value the program has to be executed again and all the values have to be entered again. This would be especially annoying for a user who had entered several values correctly only to make a small error with a later value.

In these situations it would be far preferable to have a more complex dialogue where the user is still informed of the error but is then invited to re-enter an acceptable value. This dialogue might resemble:

```
Enter a number between 1 and 10: 20
Error: Number out of range. Try again: 5
Thank you. The number you entered was 5.
```

This is a much more friendly example of interaction than those that we have been able to achieve up to now.

To implement a dialogue such as this, one or more statements in a program must be executed a number of times. Since it is not possible to predict how many times the user will make a mistake, or even if they will make one at all, the statements cannot be repeated in the program; rather, the same statements must be executed until the user provides an acceptable value.

This is only one example of where a program *loop* might be used. There are many other situations where this is required, but there are in general only three types of loop:

- Part of a program is executed a given number of times: *Do this six times.*
- Part of a program is executed until some condition becomes true: *Do this until the user has entered an acceptable value.*
- Part of a program is executed while some condition is true: *Do this while the user continues to enter incorrect values.*

In each of these examples "part of a program" means one or more statements enclosed between the usual pair of braces.

The last two possibilities in this list may seem at first sight to be the same thing, but they are not. There is a subtle difference. A loop that is executed until some condition becomes true will always be executed at least once, while a statement that is executed as long as some condition is true may never be executed at all if the condition is initially false.

Some examples of loops are:

- Execute this statement seven times.
- Execute this statement, and carry on executing it until the user enters a negative value at the prompt.
- Execute this statement while the user continues to provide valid input values.

These three types of loop are implemented in C++ as `for` loops, `do ... while` loops and `while` loops respectively. The names simply refer to the C++ keywords that are used to express such a loop.

for loops

A `for` loop causes statements to be executed a certain number of times. This number might be a known literal value such as 10, or it might be held in a variable. The first case is used when it is known that the loop will always be

executed the same number of times, while the second allows the number to be determined as the program is running.

The loop is controlled by a statement of the form:

for (<initialisation>; <continuation>; <change>)

The initialisation statement is executed the first time the loop is executed and the loop continues to execute as long as the continuation condition is true. Each time the end of the loop is reached the change statement is also executed.

This might all seem a little complicated, so an example illustrates:

```
for (int i = 0; i < 10; i ++)
```

The initialisation statement is often a declaration. In this example, when the loop is first executed an integer variable *i* will be created and initialised to 0. The loop will execute as long as *i* is less then 10 (the continuation condition) and at the end of each execution 1 is added to *i* (the change statement). The practical result of this is that in this case the loop will execute 10 times as *i* moves from 0 to 9 inclusive. It is usual to start loops with an initial value of 0[1].

The statements controlled by the loop are enclosed in the usual braces:

```
for (<initialisation>; <continuation>; <change>) {
  // statements
}
```

As usual they are indented to emphasise the fact that they are inside the loop. Any C++ statements may appear inside the loop, including conditional statements and other loops.

The variable declared in the initialisation is called the loop's *control variable*. It can be of any data type, but it is most usually an integer. It does not necessarily have to be declared inside the loop, but it most usually is.

The control variable is often used inside the loop. For example, to print the first 10 integers (starting at 0, of course) the loop would be:

```
for (int i = 0; i < 10; i ++) {
  cout << i << endl;
}
```

A slightly more useful loop prints out the 12 times table:

```
for (int i = 0; i < 13; i ++) {
  cout << i << " x 12 = " << i * 12 << endl;
}
```

This loop generates a lot of output for a small amount of code. This is the power of loops.

While the control variable can be used in any statement inside the loop, there are two important rules:

---

1 Remember that computers start to count at 0, not 1!

1. The control variable should never be altered inside the loop. It should be altered only when execution of the statements inside the loop has ended.
2. If the control variable is declared in the header of the loop it cannot be used once the loop has terminated.

If it is necessary to use the value of the control variable after the loop has finished it can be declared before the loop:

```
int i;

for (i = 0; i < 10; i ++) {
  cout << i << endl;
}
// i can be used here
```

Otherwise, attempting to use the control variable after the loop has finished will result in an error:

```
for (int i = 0; i < 10; i ++) {
  cout << i << endl;
}
// using i here is an error
```

By far the most common use of a `for` loop is just to execute a set of statements a given number of times; the easiest way of doing this is always to count up to this number. This gives a loop of a general form that will cause statements to be executed *n* times:

```
for (int i = 0; i < n; i ++) {
  // statements executed n times
}
```

Other less common possibilities include printing out just even numbers:

```
for (int e = 0; e < 10; e += 2) {
  cout << e << endl;
}
```

or counting backwards from a number:

```
for (int b = 10; b >= 0; b --) {
  cout << b << endl;
}
```

You might want to type these loops into small programs to see what they do.

`for` loops are called *determinate* loops because the number of times that the loop will be executed can always be predicted (hence determined) before the loop begins. This does not mean that the number of times a certain loop executes is always the same; the number is often controlled by, or based on, the values of other variables in the program.

For example, the loop to print the 12 times table:

```
for (int i = 0; i <= 12; i ++) {
  cout << i << " x 12 = " << i * 12 << endl;
}
```

might be more useful if the user could choose how many terms of the table to see:

```
int terms;
```

```
// get the number of terms - in reality this number
// would also be validated with a conditional statement

cout << "Enter number of terms required: ";
cin >> terms;

// display the table

for (int i = 0; i < terms; i ++) {
  cout << i << " x 12 = " << i * 12 << endl;
}
```

It is not possible to forecast how many times this loop will be executed before the program is run, but the number will always be known immediately before the loop starts.

The loop could easily be extended to allow the user to pick the first value in addition to the number of terms:

```
int start;
int terms;

cout << "Enter the first value: ";
cin >> start;
cout << "Enter number of terms required: ";
cin >> terms;

// these values should be validated here, and the
// loop executed only if they are sensible!

for (int i = start; i < terms; i ++) {
  cout << i << " x 12 = " << i * 12 << endl;
}
```

Finally, some specialised users might require only alternate terms in the table:

```
for (int i = start; i <= terms; i += 2) {
  cout << i << " x 12 = " << i * 12 << endl;
}
```

`for` loops are used when the number of times that a loop will be executed can be determined for certain before the loop is executed for the first time. If this number cannot be determined then the alternative, an *indeterminate loop*, must be used instead.

`while` loops

A `while` loop is indeterminate; this means that it causes statements to be executed while some Boolean condition is true. The condition can be anything that produces a Boolean value, much like the condition of an `if` statement. A `while` loop is started with a line of the form:

while (<condition>)

and the program statement or statements affected by the loop appear immediately below enclosed with the familiar braces:

```
while (<condition>) {
  // statements
}
```

The statements are of course indented to make it clear that they are inside the loop. As with a `for` loop, any valid statements may appear inside the loop, including other loops.

The statements will continue to be executed as long as the condition in the loop is true. When the final statement in the sequence is reached control returns to the first, and so on. This means that:

- the condition must at some point become false or the loop will never terminate
- if the condition is not true the first time the loop is to be executed the statements will never be executed at all
- the programmer must be confident of the value of the condition every time the loop is reached, including the first

`while` loops are often used to make sure that the values entered by users are as expected. If the user enters an invalid value a `while` loop can be used to provide an error message and repeat the prompt until an acceptable value is provided:

```
int aNumber;

cout << "Enter a number between 1 and 10: ";
cin >> aNumber;

while (aNumber < 1 || aNumber > 10) { // out of range
  cout << "Error: Number out of range" << endl;
  cout << "Enter a number between 1 and 10: ";
  cin >> aNumber;
}
```

The condition

```
aNumber < 1 || aNumber > 10
```

defines the state when the number entered is out of the required range and therefore invalid. The loop is executed only if the first attempt to enter the number is invalid, and it will repeat until a valid number is entered:

```
Enter a number between 1 and 10: 15
Error: Number out of range
Enter a number between 1 and 10: 20
Error: Number out of range
Enter a number between 1 and 10: 7
```

This produces a much neater dialogue between the user and the program. If the user makes a mistake they have the chance to correct the error immediately rather than having to run the whole program again.

Care must be taken that the value of the condition changes at some point in the loop. For example, this loop will never terminate:

```
aNumber = 10;
while (aNumber >= 10) {
  cout << "The number is " << aNumber << endl;
  aNumber ++;
}
```

Here the value of *aNumber* will keep increasing and the value of the Boolean expression (that *aNumber* must be greater than or equal to 10) will always be true. The loop is an *infinite loop*. In some very unusual situations an infinite loop is what is required, but such situations are very rare. If you find yourself

with a program that never seems to end, it's always a good idea to check for infinite loops.

Sometimes the statements inside a `while` loop will not be executed at all. If the condition is initially false then control skips past the loop and the statements are never executed. There are times, however, when it is clear beforehand that the statements inside the loop will always need to be executed at least once. In this case there is an alternative form of indeterminate loop that can handle this situation more elegantly.

`do ... while` loops

`do ... while` loops are a specialised form of `while` loop. In fact any `do ... while` loop can always be written as a `while` loop[2]. The only advantage of this type of loop is that the resulting code can be neater. The relationship between these two types of indeterminate loop is not unlike that between the `if` statement and the `switch` statement – it is possible to get by with only one, but using both makes for neater programs.

The statements inside a `do ... while` loop are always executed at least once. The loop is once again controlled by a condition but this time the condition is tested only after the statements inside the loop have been executed; if the condition is true then the loop is executed once again. The format of the loop makes the location of the test clear:

do {
  // statements
} *while (<condition>);*

As is now very familiar, the statements inside the loop are enclosed with braces and are indented. Also as usual, the statements may include any valid C++ statements, including other loops.

`do ... while` loops may also be used to validate values entered by a user. Their use for this is reasonable since the user is always going to have to enter the value at least once, and therefore the statements in the loop will always be executed at least once.

A `do ... while` loop can be used to validate user input:

```
do {
  cout << "Enter a number between 1 and 10: ";
  cin >> aNumber;
  if (aNumber < 1 || aNumber > 10) {
    cout << "Error: Number out of range" << endl;
  }
} while (aNumber < 1 || aNumber > 10);
```

This is obviously very similar to the example using a `while` loop and there is little to choose between the two. Both include some duplication of code; the `while` loop example has two `cin` statements and this version has the condition tested twice. The two are equivalent; the choice of which to use in a particular application would come down to the usual matter of style and taste[3].

---

2 And indeed, vice versa.

3 It's interesting that I tend to prefer the `do ... while` version, but everyone I work with seems to use the `while` version. It really is just down to style and preference – programming style can be a very personal thing.

This observation applies to most cases where an indeterminate loop is used. If it is obvious that the loop will always be executed at least once then a `do ... while` loop is the correct choice, but many programs could equally be written with either form of loop.

A final warning about == and =

Finally, a word about a very common error. Very often an indeterminate loop executes as long as some variable has a particular value. This is most commonly the value of some Boolean variable:

```
bool finished = false;

while (finished == false) {
  // statements
}
```

or in general:

```
while (<variable> == <value>) {
  // statements
}
```

A very common error and careless error is to write loops of this form as:

```
while (<variable> = <value>) {
  // statements
}
```

or

```
do {
  // statements
} while (<variable> = <value>);
```

Neither of these loops will behave as expected. In fact, both will probably never terminate no matter what value the variable has[4]. The problem is that the = (is assigned) operator has been used instead of the == (equals) operator. If you find that one of your programs has a loop that is not behaving as expected this is always something to check. The correct loops are of course:

```
while (<variable> == <value>) {
  // statements
}
```

and

```
do {
  // statements
} while (<variable> == <value>);
```

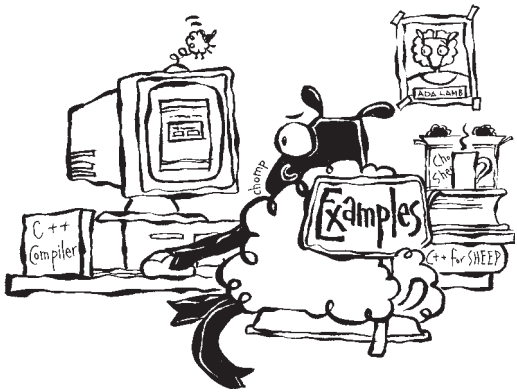
The same error can, of course, also crop up in any condition, and particularly in conditional statements.

4 Both loops would actually terminate if the value were 0.

Flow of control

Armed with loops and conditional statements you can now write programs that have a complex *flow of control*. While your programs still execute in a general sense from top to bottom you now have much more control over precisely what is going on.

This is a very basic skill of programming, so it's time to get some practice in with some examples and exercises.



Example 1 – Flossy's tables

As well as learning C++, Flossy and her friends are all brushing up their basic maths skills. To help them, Flossy has decided to write a program that will print out their times tables. Flossy plans to work out the tables up to "12 times" and the values in each up to "12 times". Both will start at 0. What sort of loop should she use?

Two loops are actually needed in this program. One will count the number of tables and one will count through the terms of the tables. Since both loops are always required to execute exactly 13 times this is a clear case where a `for` loop should be used for both.

The code seems quite complicated at first sight but it can be built up gradually.

This loop would print out some headings for each of the tables:

```
for (int table = 0; table < 13; table ++) {
  cout << "The " << table << " Times Table" << endl;
}
```

The "< 13" here is a matter of preference. Since the loop is intended to count up to 12 you might prefer:

```
for (int table = 0; table <= 12; table ++) {
  cout << "The " << table << " Times Table" << endl;
}
```

which amounts to exactly the same thing.

A different loop could be used to print out a particular table. This is the 7 times table:

```
for (int term = 0; term < 13; term ++) {
  cout << term << " x 7 = " << term * 7 << endl;
}
```

The trick in writing Flossy's program to print out a collection of tables is to put the second loop inside the first. As a first step to doing this, this pair of loops will print out the 7 times table 13 times:

```
for (int table = 0; table < 13; table ++) {
  cout << "The " << table << " Times Table" << endl;
    for (int term = 0; term < 13; term ++) {
      cout << term << " x 7 = " << term * 7 << endl;
    }
}
```

Again the indentation clearly shows that the inner loop is inside the other. The final fix is to make the number of the table that is to be generated change for each execution of the inner loop. This is easily done since the number of the required table is actually the control variable of the outer loop, *table*.

This gives the final loops in this complete program:

```
// tables.cc
//
// Program to generate and display a set of
// "times tables".
//
// Chapter 14, Examples 1.
//
// Author:    AMJ
// Date:      30th October 2001
// Platform:  g++, Linux

#include <iostream>

using namespace std;

int main ()
{

  for (int table = 0; table < 13; table ++) {
    cout << "The " << table << " Times Table" << endl;
    for (int i = 0; i < 13; i ++) {
      cout << i << " x " << table << " = "
           << i * table << endl;
    }
    cout << endl;
  }

  return 0;

}
```

Example 2 – Flossy's calculator

Flossy has many demands on her time. She is in charge of distributing the chocolate bars to the sheep and also has to spend a lot of time keeping track of her pocket money. She has decided that she needs a simple calculator to help in these important tasks. Unfortunately her pocket money does not stretch to buying a calculator, so she has decided to write one in C++.

She would like to be able to add, subtract, multiply, and divide. Since money is involved she needs to be able to cope with floating-point numbers. The calculator should prompt her for two numbers and the required operation and then display the

result. Flossy would prefer not to have to keep re-running the program for each calculation.

The code to carry out the arithmetic in this example is quite simple. For example, to carry out an addition calculation:

```
float firstNumber;
float secondNumber;

cout << "Enter the first number: ";
cin >> firstNumber;

cout << "Enter the second number: ";
cin >> secondNumber;

cout << firstNumber << " + " << secondNumber
     << " = " << firstNumber + secondNumber << endl;
```

This is fine for a program that is just going to handle addition, but a more general approach is needed for this program. At the start of the program it's quite impossible to forecast whether the calculation required will be an addition, subtraction, multiplication or division. A conditional statement is needed; `switch` is the obvious choice, as there are four distinct possible operations that might be carried out. The user must be prompted to enter the symbol representing the required calculation and a `switch` statement will choose the appropriate action and carry it out. A new variable to store the result will also be needed.

The code for a basic calculation now becomes:

```
float firstNumber;
float secondNumber;
float result;
char operation;

cout << "Enter the first number: ";
cin >> firstNumber;

cout << "Enter the second number: ";
cin >> secondNumber;

cout << "Enter the operation (+, -, *, /): ";
cin >> operation;

switch (operation) {
  case '+': result = firstNumber + secondNumber;
            break;
  case '-': result = firstNumber - secondNumber;
            break;
  case '*': result = firstNumber * secondNumber;
            break;
  case '/': result = firstNumber / secondNumber;
            break;
}

cout << firstNumber << ' ' << operation << ' '
     << secondNumber << " = " << result << endl;
```

As it stands this `switch` statement will fail and incorrect results will be produced if an invalid character (one that does not represent an operation) is entered. This means that the operation character (stored in the variable called *operation*) must be validated as it is entered. This involves using a suitable loop to warn the user of the error and to ask them to re-enter. The code to do this follows a familiar pattern[5]:

```
cout << "Enter the operation (+, -, *, /): ";
cin >> operation;

while (operation != '+' && operation != '-'
       && operation != '*' && operation != '/') {
  cout << "Error: Invalid Operation" << endl;
  cout << "Enter the operation (+, -, *, /): ";
  cin >> operation;
}
```

This ensures that the `switch` statement has a valid character to process.

Flossy wants the program to execute over and over again, so obviously another loop, controlling the whole program, is required. This loop will always execute at least once, so this time a `do ... while` loop is the appropriate choice. The loop is controlled by a Boolean variable *finished*, which is initially false and is set to true only when the user indicates that they do not want to run the program again.

A prompt is needed to ask the user if they want to run the program again and a simple dialogue gets the user's response and stores it in a `char` variable *answer*:

```
cout << "Run Again? (y/n)? ";
cin >> answer;
```

and a conditional statement sets the *finished* variable if appropriate:

```
if (answer == 'y') {
  finished = true;
}
```

or, equally:

```
finished = (answer == 'y');
```

Slightly better programming style (and dialogue design) is to consider the fact that the user might enter either an upper- or lower-case letter and to use:

```
if (answer == 'y' || answer == 'Y')
```

as the conditional statement.

If a program is to present the user with a good dialogue it should also enforce the entry of *y* or *n* as an answer to this prompt. This extra feature has been added to the complete program that follows. This program also handles the case where the operation is division and the second number is 0; it is necessary to avoid an error caused by a division by this 0.

---

5 You have now seen a lot of familiar patterns as you have read these chapters. This is a good thing. One of the "tricks" of programming is spotting these patterns in a problem, and realising that a solution to some other problem can be adapted to the new problem. This is why it's a good plan to make sure that you never throw away any programs that you write.

Here is the final program. While the statements that carry out the steps to solve the program are very few in number, there are many more that control the program flow, produce the dialogue and deal with possible error cases. It is quite usual for a great deal of any program to carry out tasks such as these.

Take a close look at this program and make sure that you understand what's going on.

```
// calculator.cc
//
// A simple calculator for all Flossy's
//   calculating needs.
//
// Chapter 14, Examples 2.
//
// Author:    AMJ
// Date:      30th October 2001
// Platform:  g++, Linux

#include <iostream>

using namespace std;

int main ()
{
  bool finished = false;
  char answer;

  float firstNumber, secondNumber;
  char operation;
  float result;

  while (!finished) {

    // Get Input Values

    cout << "Enter the first number: ";
    cin >> firstNumber;

    cout << "Enter the second number: ";
    cin >> secondNumber;

    cout << "Enter the operation (+, -, *, /): ";
    cin >> operation;

    while (operation != '+' && operation != '-'
           && operation != '*' && operation != '/') {
      cout << "Error: Invalid Operation" << endl;
      cout << "Enter the operation (+, -, *, /): ";
      cin >> operation;
    }

    // Calculations

    if (operation == '/' && secondNumber == 0) {
      cout << "Error: Cannot Divide by Zero" << endl;
    }
    else {
      switch (operation) {
```

```
            case '+': result = firstNumber + secondNumber;
                      break;
            case '-': result = firstNumber - secondNumber;
                      break;
            case '*': result = firstNumber * secondNumber;
                      break;
            case '/': result = firstNumber / secondNumber;
                      break;
          }

          // Display Result

          cout << firstNumber << ' ' << operation << ' '
               << secondNumber << " = " << result << endl;

        } // else

        // Running again?

        cout << "Run Again? (y/n)? ";
        cin >> answer;

        do {

          if (answer == 'n' || answer == 'N') {
            finished = true;
          }
          else if (answer != 'y' && answer != 'Y') {
            cout << "Error: Please enter \'y\' or \'n\'"
                 << endl;
            cout << "Run Again? (y/n)? ";
            cin >> answer;
          } // else
        } while (answer != 'y' && answer != 'n'
                 && answer != 'Y' && answer != 'N');

      } // while !finished

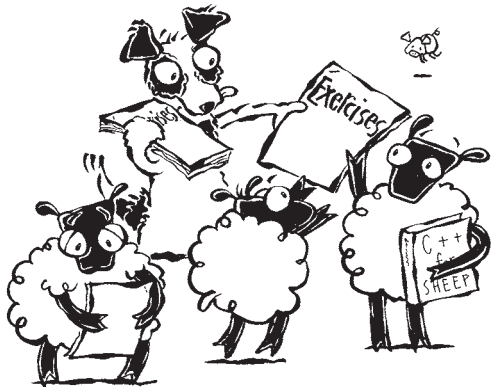
      return 0;

    }
```

This is the longest program that you have seen so far and there are a few things to notice:

- The indentation shows at all times which statements are controlled by which control statements.
- There are often loops within loops and conditional statements within other conditional statements and loops. The indentation helps to keep track of what is inside what else.
- There are sometimes many statements enclosed between one set of braces. Where the closing brace is a long way away from the corresponding opening brace a brief comment has been added to emphasise what the brace is ending.

This is a lengthy program containing most of the C++ that you have met so far. Take some time to go through it and make sure you understand it.



14.1 Write a program that will print a triangle made up of stars, as follows:

```
*
**
***
****
*****
```

The number of stars in the final line should be provided by the user in answer to a suitable prompt. The program should ensure that the provided value is at least one and should allow the user to re-enter the value if an error is made.

14.2 Write a similar program that prints a square made up of stars:

```
********
********
********
********
********
********
********
********
```

As before, the number of stars making up each side of the square should be provided by the user. This number should be positive.

14.3 Using your two programs as a starting point write a program that presents the user with a small menu:

```
Shape Menu
~~~~~~~~~~
1. Draw a Triangle
2. Draw a Square
Q. Quit
```

The user should select an option from the menu and should be prompted to enter the required size of the shape. The program should continue to execute until the user chooses the Quit option. If the option chosen is not valid, an error message should be displayed and then the menu should be shown again.

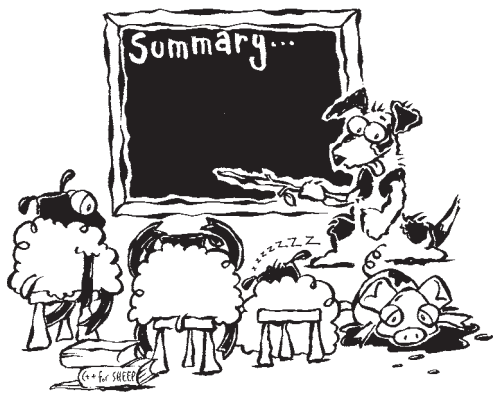
14.4 Add a further shape to your program. A rectangle requires a height and a width to be entered:

```
******
******
******
```

14.5 Finally change your program so that the user can choose the character that is used to draw the shapes. This should be done by adding an option to the menu:

```
C. Change Character
```

The first time the program is executed the character should still be *. The character used must be one of $, %, * or ^.

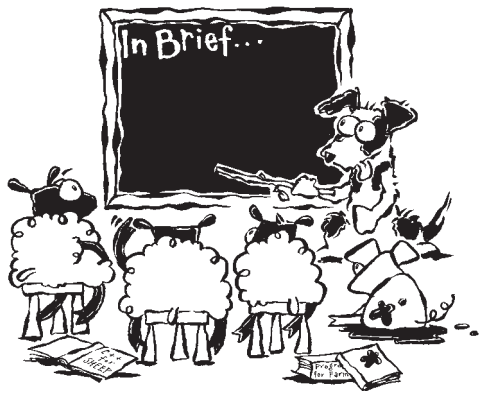


Program loops allow a great deal of flexibility in managing the way that control flows in a program. They allow for the development of much more sophisticated dialogues and mean that programs may continue to execute until the user has finished with them. Together with conditional statements, loops give a programmer full control over the order in which program statements are executed.

There are two general types of loop. The first, called determinate loops, are used when the number of times that a loop will execute is known before it is started. The alternative, called indeterminate loops, give the extra flexibility of executing a loop under the control of some Boolean condition. There are two forms of indeterminate loop, one providing for the special case where it is known that a loop will be executed at least once.

Loops and other control statements can be used in any program and also in methods. It is now time to see how they can be used to write more complicated and sophisticated methods in a class.


Chapter 15
More Methods
FIRST
AID KIT
Methods
Attributes of Part #XYZ



The techniques for controlling the flow of execution of statements that you have seen and practised in the previous two chapters can also be used in methods. Up to this point the methods you have written have been very simple; most of them have done nothing more complicated than return a single value after perhaps doing some simple calculation. Now you should be able to use the techniques from the last two chapters to write methods that carry out more complicated processing.

This chapter explains how to write more complex methods. Much of this involves nothing more complicated than using conditional statements and loops to develop more sophisticated methods. There is only one new bit of C++ in this chapter; sometimes methods are required that can return more than one value and this chapter shows you how to write methods that do that.

After reading this chapter you should be able to implement most if not all of the methods needed for your classes. You should be able to write methods that make use of conditional statements and loops, and also methods that return more than a single value. Of course, you should also be able to use these methods from your C++ programs.



There is not much new C++ in this chapter. It should be fairly clear how to use conditional statements and loops in methods, so most of this chapter will

just provide you with some examples. First though, here is a recap of the meaning of a C++ method description:

<return type> <method name> (<parameter types>);

The description starts with the type of value that is returned by the method. This is followed by the name of the method, which is in turn followed by the types of any parameters, in brackets and separated by commas. The line ends with a semi-colon. If a method is guaranteed not to change the values of any of the private attributes in the class, the word `const` is added before the semi-colon and after the closing bracket.

The correct name for such a description is a *prototype*. The description has in fact much the same function as a prototype in more familiar engineering applications where, for example, a model of a car is made before the car itself. A method's prototype shows a potential user exactly what the method will do when it is written. Armed with the information gleaned from the prototype a programmer can write a program that will make use of the method without having any knowledge at all of how the method achieves its result. Also, a compiler can compile the part of the program that uses the method with no knowledge of how the method works; the method itself can be linked in later.

All methods determine and return exactly one value of the type given as the first part of the prototype as the return type; if a method does not return a useful value at all it is defined to return one of the `void` type. Methods sometimes require values to process in order to produce their result; these are the method's parameters, the types of which are listed in brackets after the name of the method. The final brackets are included but left empty if there are no parameters.

For example the prototype:

```
float distanceFrom (int, int) const;
```

defines a method called *`distanceFrom`* that returns a single floating-point value and requires two integer parameters. The final `const` at the end of the line indicates that this method when called does not change any of the values of the private attributes of the class with which it is associated.

Sometimes a method is required that will return more than one value. In the examples you have used so far this has not been possible; there is no mechanism at all to do this. This can be achieved in C++ by using a subtly different kind of parameter. There are in fact two kinds of parameter that can be provided for methods: *value parameters* and *reference parameters*.

Value parameters

The parameters in all the methods you have seen, used and written so far have been *value parameters*. These are parameters that simply provide values for the methods to use in their processing; the values have probably come from the program that was making use of the methods. The value may be a literal value (1, 2.5 or some string) or the value held in a variable.

For example, in a *`Sheep`* class this method:

```
void printDiscountValue (float) const;
```

might be implemented to print out the value of the sheep (from a private attribute *`value`*) after some discount rate has been applied; the rate is provided as a parameter. Its implementation would be:

```
void printDiscountValue (float discount) const
{
  cout << value - (value * discount);
  return;
}
```

and it would be called:

```
Sheep freda;
float rate = 0.10;

freda.setValue (100.0);

cout << "Price less discount: "
     << freda.printDiscountValue (rate) << endl;
```

When the method is called the value held in the variable *rate* is passed into the method and is copied into the parameter *discount*. This value is then used in the method. When the method has finished control returns to the calling program and the *discount* variable no longer exists. *discount* is said to have been a *local variable* in the method.

Suppose that the discounted price were worked out in a more complicated way. If all discounts were changed so that 10% more than the discount were subtracted from the value the method would be changed to:

```
void printDiscountValue (float discount) const
{
  discount += 0.10;
  cout << value - (value * discount);
  return;
}
```

The program using the method would be exactly the same. After the call to the method the *discount* variable would be lost and the program would continue. The variable *rate* in the calling program would, of course, have the same value as before. In this method *rate* is a value parameter; it just provides a value that the method processes.

This raises the question of how a method would change the value of a variable in a program that used the method. This is precisely what can be achieved with reference parameters.

Reference parameters

The key to writing methods that can return more than one value is to use a different kind of parameter. Value parameters only allow values to be passed *into* a function; a mechanism is needed where values can also be passed back *out of* a method. C++ provides a type of parameter that will do this in *reference parameters*. In some other languages reference parameters are called *variable parameters*. Even though the names are different, the way of using them is exactly the same. For reasons that will become clear, reference parameters are always variables; they cannot be literal values.

At first, reference parameters behave in exactly the same way as value parameters. The value held in the variable being used as the parameter is passed into the method and can then be used as the method executes. But what has been passed is not just the value of the variable: it is a reference to the variable itself. The leads to the crucial difference between the two types of parameter, which becomes apparent in what happens when the method

has finished executing. While the final value of a variable passed as a value parameter is lost, the final value of a variable passed as a reference parameter is still in the corresponding variable (the reference) in the main program. This does indeed allow more than one value to be returned from a method.

This probably seems quite complicated, but it really is very straightforward. Here's some syntax and a simple example.

A reference parameter is declared in the same way as a value parameter except that an & symbol is added in the prototype:

```
void addThisSheepValue (float&) const; // reference parameter
```

This (admittedly a little contrived) method is intended to add the value of an instance of the *Sheep* class to some running total. Assuming the obvious function of the other methods and private attributes it would be called like this:

```
Sheep flossy;
Sheep freda;

float totalValue = 0.0;

flossy.setValue (110.0);
freda.setValue (120.0);

flossy.addThisSheepsValue (totalValue);
freda.addThisSheepsValue (totalValue);
```

and the implementation is straightforward:

```
void Sheep::addThisSheepsValue (float& runningTotal) const
{
  runningTotal += value;
  return;
}
```

Each time the method is called it is not the value of *totalValue* from the main function that is copied into *runningTotal* (which exists only inside the method), it is a reference to the variable itself. This means that *totalValue* and *runningTotal* represent the same variable – they actually correspond to the same location in the computer's memory. When the value of *runningTotal* is changed in the computer's memory, the value of *totalValue* is also changed, and this will hold the final value when the method has finished. The value of *totalValue* changes like this:

```
Sheep flossy;
Sheep freda;

float totalValue = 0.0;

flossy.setValue (110.0);
freda.setValue (120.0);

cout << totalValue << endl; // totalValue is 0.0

flossy.addThisSheepsValue (totalValue);
cout << totalValue << endl; // totalValue is 110.0

freda.addThisSheepsValue (totalValue);
cout << totalValue << endl; // totalValue is 230.0
```

If a method has two reference parameters then it may return more than one value. Suppose a method were required that returned a sheep's value and also an indication of whether the sheep was aged more than two years. One way of doing this would be to use the return value of the method for one value and a reference parameter for the other:

```
bool Sheep::getValueAndOverTwo (float& v) const
{
  v = value;
  return age > 2;
}
```

This is entirely legal C++ and would work, but it is not especially elegant. Also there seems no particularly good reason why one of the two values should be a parameter and the other should be returned. This seems to make an unnecessary and non-existent distinction between the two values being returned. The call to this method might also be confusing:

```
float value;
bool old;

Sheep flossy;

old = flossy.getValueAndOverTwo (value);
```

The call seems to imply that the outcome of the method, in terms of the value that is assigned to *old*, depends in some way on the value. This is not the case.

A far better solution is to use two reference parameters and to have the method return no value (`void`):

```
void getValueAndOverTwo (float& v, bool& o) const
{
  v = value;
  o = age > 2;

  return;
}
```

The call to the method is much clearer:

```
flossy.getValueAndOverTwo (value, old);
```

As usual when there is more than one parameter they must be supplied in a comma-separated list and they must be supplied in the correct order. And as before, the identifiers of the variables used as parameters do not have to match the identifiers used in the implementation of the method; in many cases the programmer would not even have any idea of what these are.

Behind the scenes

It is probably easier to understand the difference between the two kinds of parameters by thinking about what is happening when the values are passed. A value parameter passes just that, a value. A reference parameter is different; it passes a reference to a variable. In practical terms this means that a reference parameter actually passes the location of the variable in the computer's memory. The method does not create its own copy of a reference parameter; it uses the same piece of memory for its version of the variable, so

any changes to the local value in the method change the value in the calling program too.

It follows that a reference parameter must always be a variable, whereas a value parameter can be a variable or an explicit (literal) value. For example, the method:

```
float Sheep::getDiscountPrice (float&);
```

can be called only with a variable as a parameter and never with a value:

```
float rate = 0.10;

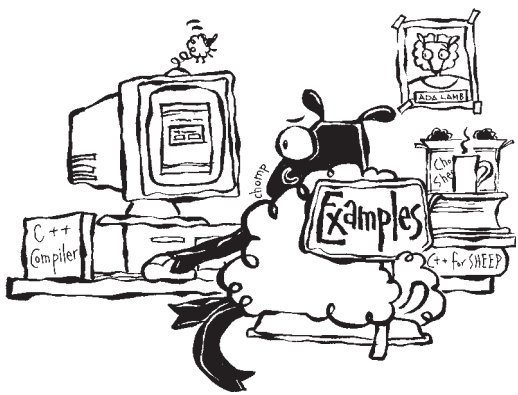
flossy.getDiscountPrice (rate); // this is correct
flossy.getDiscountPrice (0.10); // this is an error
```

The reason for this is obvious. Once the method has executed the final value of the parameter is still available in the variable. If there is no variable, only a value, there is nowhere for this value to be copied.

Parameters and control

There has been only one new piece of C++ in this chapter, but it is an important one. Many new programmers find reference parameters a difficult concept to grasp. It is important that you get some practice and that you think about the difference between the two types of parameter. When deciding which to use, the question to ask is always whether the final value of the variable from the method is needed in the calling program.

It is also time to practise writing some more complicated methods using conditional statements and loops. There has been nothing at all about doing this in this chapter because you already know how to do it! All you need is some practice, so it's on to the examples.



Example 1 – Locating sheep

The example `Sheep` *class in Chapter 8 included a method intended to determine whether a sheep was at a certain position. Its prototype was:*

```
bool atPosition (int, int) const;
```

What would be the implementation of this method?

The position of the sheep is stored in two private attributes, called just *x* and *y*. The first value provided to this method is the *x* value to be checked and the

second is the *y* value. These parameters might be called *xPos* and *yPos* in the method to make their meaning clear. If *x* is the same as *xPos* and *y* the same as *yPos*, then the sheep is at the location and the method should return `true`; otherwise the method should return `false`.

A simple conditional statement is needed to check whether or not the sheep is at the particular point. The implementation is:

```
bool atPosition (int xPos, int yPos) const
{
  if (x == xPos && y == yPos) {
    return true;
  }
  else {
    return false;
  }
}
```

This structure in a method is very common. It states that if a certain condition is true the method returns `true` and otherwise it returns `false`. It is worth remembering that such a method can also be implemented by simply returning the condition; this example could equally well be written:

```
bool atPosition (int xPos, int yPos) const
{
  return x == xPos && y == yPos;
}
```

This second form is obviously much shorter, but some might find it too terse and the longer form clearer. The choice between the two is a question of style and preference.

To illustrate this point about style one last time, this function could also be written[1]:

```
bool atPosition (int xPos, int yPos) const
{
  if (x == xPos && y == yPos) {
    return true;
  }

  return false;
}
```

This version works because the second `return` will never be reached if the expression is true.

While all these three versions are equivalent, it would be good practice to choose the one that you prefer and make sure that you always use the same one; good style is consistent.

Example 2 – Locating sheep 2

The Chapter 8 example also contained a method with the prototype:

```
float distanceFrom (int, int) const;
```

---

1 My friend Karim gave me this version when he was checking the C++ in this chapter for me. He would always write methods like this in this way. I never would; I'd use the version that just returns the condition. Style is like that; both approaches are fine as long as we're both consistent.

This method finds the distance that a sheep is from the provided point. What is its implementation?

This calculation requires a small amount of mathematics, but do not be alarmed!

The calculation involves finding the difference between the two *x* values and the two *y* values and then using Pythagoras' theorem to find the distance. Pythagoras' theorem involves using a square root; C++ provides the `sqrt` function to find square roots. `sqrt` is actually a C function and requires that the maths library is included:

```
#include <cmath>
```

The implementation is:

```
float distanceFrom (int xPos, int yPos) const
{
  int dx = xPos - x;
  int dy = ypos - y;

  return sqrt (float (dx * dx + dy * dy));
}
```

Since `sqrt` works only on floating-point values[2], a cast is needed. Including it in a program may mean that you have to tell the compiler to link new libraries; it's time to check the *Local guide*.

Example 3 – The Big Hole

Farmer Strawstrap is concerned about a large hole that has appeared in the field where Flossy and her friends live. He is especially worried that one of the (valuable) sheep may fall down the hole and so sustain an injury that would reduce its value.

He would like a program that will track each of his sheep and will tell him how far away they are from the hole. The program should also warn him if a sheep falls down the hole.

In order to test the program Flossy has been fitted with a tracking device. The field is marked out in units of one metre, and the hole is in the field at coordinates 5,3.

The program should prompt the farmer to enter the direction in which the tracking device indicates that Flossy has moved. It should then display her distance from the hole and should print a warning if she is closer than two metres from it. It should provide another warning if she actually falls down.

What class would be needed for this program? What does the program look like?

A *Sheep* class similar to that in Chapter 8 will be needed for this program. There will need to be two private attributes: one for the *x* position and one for the *y* position. It will also be handy to store the name of the sheep. The following methods will also be needed:

```
void moveNorth ();
void moveSouth ();
void moveEast ();
void moveWest ();
```

2 Some C++ compilers will provide a version of `sqrt` that works with integers. The one I used for these programs didn't.

```
  void setX (int);
  void setY (int);
  int getX () const;
  int getY () const;

  float distanceFrom (int, int) const;
  bool atPosition (int, int) const;
```

The implementation of these methods is straightforward. The *distanceFrom* method is the same as in the previous example, and *atPosition* is as in the first example in this chapter.

The program is on the whole straightforward, but it's worth going through it since it contains almost all the C++ we've seen so far:

- Conditional statements are used to check the input values.
- Conditional statements are used to check the sheep's distance from the Big Hole.
- Loops are used to run the whole program until the user has finished.
- Loops are used to request that the user re-enters incorrect values.

The complete classes and program are below. It is certainly worth spending some time going through them all to make sure that you understand what's going on. If you can, get the program off the web site and play around with it.

```
// sheep.h
//
// Definition of a simple Sheep class.
// Includes methods to deal with the "Big Hole" problem.
//
// Chapter 15, Examples 3.
//
// Author:    AMJ
// Date:      8th December 2001
// Platform:  g++, Linux

#include <string>

using namespace std;

class Sheep {
  private:
    string name;
    int x, y;

  public:
    Sheep (); // sets no default values

    // Selectors
    string getName () const;
    int getX () const;
    int getY () const;

    // Mutators
    void setName (string);
    void setX (int);
    void setY (int);

    // Moving
```

```
    void moveNorth ();
    void moveSouth ();
    void moveEast ();
    void moveWest ();

    // Distance from a Point

    float distanceFrom (int, int) const;

    // At a location?

    bool atPosition (int, int) const;
};

// sheep.cc
//
// Implementation of a simple Sheep class.
// Includes methods to deal with the "Big Hole" problem.
//
// Chapter 15, Examples 3.
//
// Author:    AMJ
// Date:      8th December 2001
// Platform:  g++, Linux

#include <cmath> // for sqrt ()
#include <string>

#include "sheep.h"

using namespace std;

Sheep::Sheep ()
{
  return;
}

// Selectors

string Sheep::getName () const
{
  return name;
}

int Sheep::getX () const
{
  return x;
}

int Sheep::getY () const
{
  return y;
}

// Mutators

void Sheep::setName (string newName)
{
  name = newName;
```

```
  return;
}

void Sheep::setX (int newX)
{
  x = newX;
  return;
}

void Sheep::setY (int newY)
{
  y = newY;
  return;
}

// Moving

void Sheep::moveNorth ()
{
  x ++;
  return;
}

void Sheep::moveSouth ()
{
  x --;
  return;
}

void Sheep::moveEast ()
{
  y ++;
  return;
}

void Sheep::moveWest ()
{
  y --;
  return;
}

// Distance from a Point

float Sheep::distanceFrom (int xPos, int yPos) const
{
  int dx = xPos - x;
  int dy = yPos - y;

  return sqrt (float (dx * dx + dy * dy));
}

// At a location?

bool Sheep::atPosition (int xPos, int yPos) const
{
  return x == xPos && y == yPos;
}
```

```
// bigHole.cc
//
// Farmer Strawstrap and the Big Hole.
//
// Chapter 15, Examples 3.
//
// Author:    AMJ
// Date:      8th December 2001
// Platform:  g++, Linux

#include <iostream>
#include <string>

#include "sheep.h"

using namespace std;

int main ()
{

  Sheep aSheep;

  string name;
  int x, y;

  const int BigHoleX = 5;
  const int BigHoleY = 3;

  char direction;
  char answer;

  bool finished = false;

  // Initial Details

  cout << "Enter the name of the sheep to be tracked: ";
  cin >> name;
  cout << "Enter " << name << "\'s current x position: ";
  cin >> x;
  cout << "Enter " << name << "\'s current y position: ";
  cin >> y;

  aSheep.setName (name);
  aSheep.setX (x);
  aSheep.setY (y);

  // Main Loop

  while (!finished) {

    // Move the Sheep - Get Direction

    do {
      cout << "Enter " << aSheep.getName ()
           << "\'s direction (n, s, e, w): ";
      cin >> direction;

      if (direction != 'n' && direction != 'N' &&
          direction != 's' && direction != 'S' &&
```

```
        direction != 'e' && direction != 'E' &&
        direction != 'w' && direction != 'W') {

      cout << "Error: Invalid Direction!" << endl;

      cout << "Enter " << aSheep.getName ()
           << "\'s direction (n, s, e, w): ";
      cin >> direction;
    }
  } while
    (direction != 'n' && direction != 'N' &&
     direction != 's' && direction != 'S' &&
     direction != 'e' && direction != 'E' &&
     direction != 'w' && direction != 'W');

  // Now Move the Sheep

  switch (direction) {
    case 'n':
    case 'N': aSheep.moveNorth ();
              break;
    case 's':
    case 'S': aSheep.moveSouth ();
              break;
    case 'e':
    case 'E': aSheep.moveEast ();
              break;
    case 'w':
    case 'W': aSheep.moveWest ();
              break;
  }

  // Report New Position and Other Facts

  cout << aSheep.getName () << " is now at ("
    << aSheep.getX () << ", " << aSheep.getY ()
    << ")." << endl
    << "This is "
    << aSheep.distanceFrom (BigHoleX, BigHoleY)
    << " metres from the Big Hole." << endl;

  // Check Warnings

  if (aSheep.distanceFrom (BigHoleX, BigHoleY) < 2.0) {
    cout << "*** WARNING *** " << endl
         << aSheep.getName ()
         << " is dangerously near the hole."
         << endl;
  }

  if (aSheep.atPosition (BigHoleX, BigHoleY)) {
    cout << "*** WARNING *** " << endl
         << aSheep.getName ()
         << " has fallen in the hole."
         << endl;
  }
```

```
    // Run again?

    do {
      cout << "Carry on? ";
      cin >> answer;

      if (answer != 'n' && answer != 'N'
          && answer != 'y' && answer != 'Y') {
        cout << "Error: Enter 'y' or 'n'!" << endl;
        cout << "Carry on? ";
        cin >> answer;
      }
    } while (answer != 'n' && answer != 'N'
             && answer != 'y' && answer != 'Y');
    finished = answer == 'n' || answer == 'N';

  } // Main Loop

  // All Done

  return 0;
}
```

Example 4 – The complexity of sheep values

Farmer Strawstrap remains concerned about the accuracy of his strategy for retrieving the values of this sheep. His current program has a method:

```
void getValue () const;
```

which simply returns the value of a sheep based on a private attribute (and some incredibly complicated formulas).

Unfortunately there have been some cases where this figure has proved to be woefully inaccurate. The price received for a sheep at the market has often been less than the actual value, but it has sometimes been more. The farmer decides to add a method that will return two values to represent the best and worst possible values for the sheep. He believes that the best possible value is 10% more than the actual value and the worst possible is 25% less.

What is the prototype for this method? What is its implementation?

This method returns two values, so reference parameters will be needed. The best solution is to write the method to have two reference parameters and to return `void`:

```
void getPossibleValues (float&, float&) const;
```

The implementation simply uses the *`value`* private attribute (we will ignore any complex formulas here) and sets the two parameters to the appropriate values:

```
void Sheep::getPossibleValues (float& best, float& worst) const
{
  best = value * 1.1;
  worst = value * 0.75;
  return;
}
```

Example 5 – Old sheep are cheap

The farmer is impressed with the new method, but decides that it is still not accurate enough. While it works splendidly for young sheep, it is not correct for sheep aged

more than five. For such old sheep the worst price is 40% of the stored value and the best price is the stored value itself. The method must be changed!

This change is quite straightforward. The age of the sheep is available in a private attribute and all that is needed is a simple conditional statement. There are, however, two ways of achieving the same result!

The first possible solution uses a conditional statement to set the two percentage values, which are stored in two local variables:

```
void getPossibleValues (float& best, float& worst) const
{
  float bestRate, worstRate;

  if (age > 5) {
    bestRate = 1.0;
    worstRate = 0.4;
  }
  else {
    bestRate = 1.1;
    worstRate = 0.75;
  }

  best = value * bestRate;
  worst = value * worstRate;
  return;
}
```

An alternative uses a conditional statement to control the calculation itself:

```
void getPossibleValues (float& best, float& worst) const
{
  if (age > 5) {
    best = value * 1.0;
    worst = value * 0.4;
  }
  else {
    best = value * 1.1;
    worst = value * 0.75;
  }

  return;
}
```

This version has the small advantage of not requiring two additional variables inside the method but the choice between the two probably comes down to the programmer's preference and style. Many would argue that the more long-winded approach in the first version makes the reasons for the various steps more obvious. The first version would probably also be slightly easier to alter if the percentages had to be changed for some reason.

Example 6 – Some sheep are too cheap

The results of using this new method are somewhat alarming, and the farmer begins to realise that some of his sheep have very little value. He decides that the method for determining the possible value of a sheep should also return a warning if in the worst case a sheep is worth less than £10. How can the method be altered to achieve this?

A sheep is either worth less than £10 or it is not, so this value is Boolean. A Boolean `true` value will represent a value less than £10 and `false` will represent any other value. At present the prototype of the method is:

```
void getPossibleValue (float&, float&) const;
```

The new Boolean value can be added in one of two ways. The first is to add a new reference parameter:

```
void getPossibleValue (float&, float&, bool&) const;
```

The value is then calculated in the method:

```
void getPossibleValues (float& best, float& worst,
                        bool& tooCheap) const
{
  if (age > 5) {
    best = value * 1.0;
    worst = value * 0.4;
  }
  else {
    best = value * 1.1;
    worst = value * 0.75;
  }

  tooCheap = (worst < 10.0);

  return;
}
```

An alternative approach is to retain the same parameters and to use the currently unused return value from the method to return the Boolean:

```
bool getPossibleValues (float& best, float& worst) const
{
  if (age > 5) {
    best = value * 1.0;
    worst = value * 0.4;
  }
  else {
    best = value * 1.1;
    worst = value * 0.75;
  }

  return worst < 10.0;

}
```

These two approaches are effectively the same. There is a potentially important difference in the way they are called. Suppose that a program had created an instance of a *Sheep* class called *aSheep* and was required to call the method to determine the possible values and whether or not the sheep potentially had too low a value. Using the first method the program would be:

```
float high, low;
bool tooCheap;

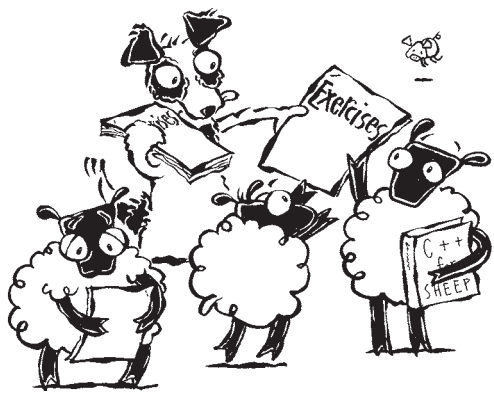
aSheep.getPossibleValues (high, low, tooCheap);
```

```
if (tooCheap) {
  cout << aSheep.getName () << " is too cheap!" << endl;
}
```

If the second version of the method has been used the call to the method would be different, and it can be combined with the conditional statement:

```
float high, low;
if (aSheep.getPossibleValues (high, low)) {
  cout << aSheep.getName () << " is too cheap!" << endl;
}
```

This version is shorter and probably neater (even if some might prefer to separate out the test of the condition), although, if it were used, the method's name should be changed to something that might give a better idea of the meaning of the return value. The choice of which of these to use would depend on the details of the program that would use the method, and to some extent also on style.



15.1 The *Sheep* class used in the last three examples is defined in the header file:

```
// sheep.h
//
// Store sheep's name and value.
//
// Chapter 15, Exercises 1.
//
// Author:    AMJ
// Date:      19th December 2001
// Platform:  g++, Linux

#include <string>

using namespace std;

class Sheep {
  private:
    string name;
    int age;
    float value;
```

```
    public:
      Sheep ();

      void setName (string);
      void setAge (int);
      void setValue (float);

      string getName () const;
      int getAge () const;
      float getValue () const;

      void getPossibleValues (float&, float&);
  };
```

Implement this class. Test the class by writing a program that prompts the user to enter the details of a sheep and then displays the highest and lowest possible values of the sheep.

Assume that the lower value is 60% of the stored value and the higher is 130%.

15.2 Modify your program so that it processes a number of sheep. After the values have been displayed the program should display a prompt:

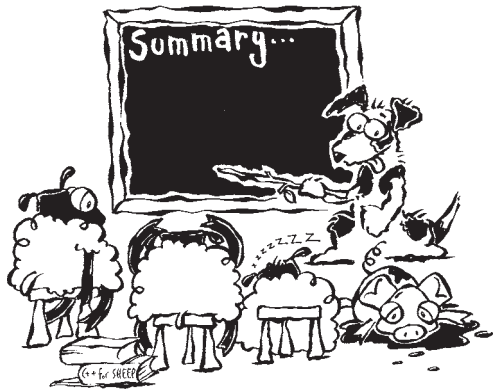
```
Another Sheep? (y/n):
```

and should run again if the user answers *y*. If the user does not enter *y* or *n* an error message should be displayed and the prompt repeated.

15.3 Modify the program again to display a warning message if the sheep is worth less than £10 in the worst case. Do not modify the methods!

15.4 Examine your *setAge ()* method. What would happen if the user entered a negative value for the age? Modify your program to reject negative values for the sheep's age. Do the same for the sheep's value.

15.5 Checks similar to those you have just added would have to be implemented each time that the *setAge ()* method was called. There is therefore a strong argument that the check should take place inside the method. Modify the method to return `true` if the age was valid and `false` otherwise. If the age was invalid the private attribute should be set to 0. Do the same for the *setValue ()* method. You will need to change both prototypes (in particular the return type) and implementations. Finally, alter your test program to use the new methods.



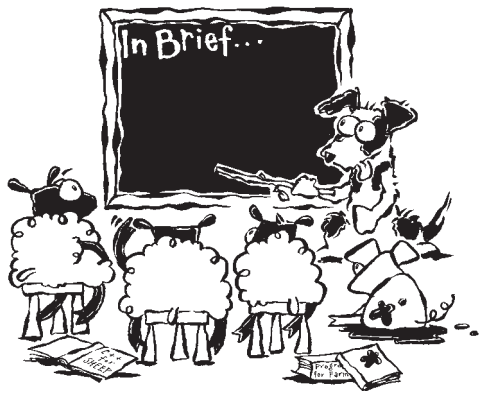
Methods can have two different kinds of parameter. The simplest case is the value parameter that simply provides a value for the method to process; the value may come from a variable or may be explicitly included. The slightly more complicated situation is a reference parameter. A reference parameter is always a variable; a reference to the variable is passed into the method, which can make use of the variable's initial value. The same location in the computer's memory is used in the method as in the main program, so any changes made to the value in the method also affect the corresponding variable in the main program.

Value parameters allow values to be passed into a method. Reference parameters extend this to allow values to be passed out of a method, or into and out of a method. The syntax for denoting the type of parameter is to add an & character after the type of a reference parameter.

Any valid C++ may be used in a method, including conditional statements and loops. You should now be able to use all the C++ that you have learned so far and you should be able to write complex methods and classes.

As you write more complicated classes your programs will also become more complex and longer. You may well find that a program of several hundred lines is quite difficult to edit and manage; it can be awkward to find the particular line that is causing an error. C++ provides a neat mechanism to split up long programs into smaller units that look not unlike methods; these units, called *functions*, are described in the next chapter.


Chapter 16
Making things Happen.
In Bite Size Chunks.
RoTa tHiS Week
Flossy
Freda
Peter
Clare
Mucking Out
PLANNING
Grass Cropping
Frolicking



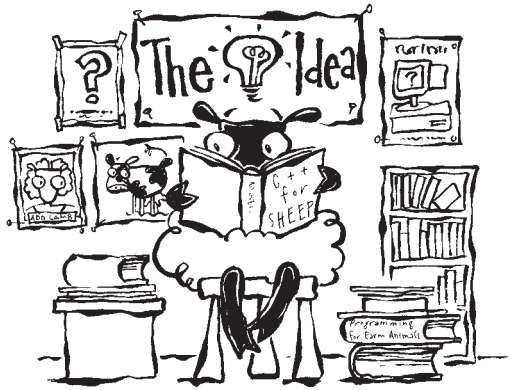
By now some of the programs that you have seen and written are probably getting quite long. It's more than likely that together with the implementations and definitions of classes some of your programs will now be several hundred lines long; the "Big Hole" example in the previous chapter amounts to almost 300 lines. While this is still small when compared to many commercial programs (which will often consist of many hundreds of thousands of lines of code), you are probably finding that the programs are starting to become unwieldy. You may well have had problems scrolling up and down large files trying to find a particular part of a program or a particular error. It is not sensible to store programs of any length in a single file; imagine trying to find a small error in a single file containing many hundreds of thousands of lines.

In general, programmers find it easier to work with small parts of a program. When they are finished these small parts can later be put together and combined into the final program. C++ provides a way of doing this: programs can be broken down into *functions*. This chapter describes how to implement and use functions in your programs.

The use of functions to break down a program into easy-to-handle small units is not unlike the process of breaking down a solution to a problem into classes and then a program. The key idea is that if the program that a programmer has to develop is kept as small as possible, the more likely it is to be correct and the easier it will be to write and test.

Some tasks are needed in many programs and are so common that C++ provides built-in ready-written and tested functions to carry them out. Most of these are for basic mathematical tasks (we have already met `sqrt` to find square roots), but there are also some to provide useful operations on characters and strings. Many of the built-in functions are in fact inherited from the C language, so many of them also require that C libraries are included.

After reading this chapter you should understand how to write C++ programs that make use of functions. You should understand how to use the built-in functions provided in C++ and also how to write and use your own functions. You should now be able to write complete C++ programs that make use of functions to break the overall task down into more manageable smaller units.



Up to now all the programs that you have written have very probably been written in a single file. Your classes have been written in separate files, but the program that made use of the classes has always been stored in a single file. The execution of the program has always basically flowed from the top of this file to the bottom, and this may well have been through many hundreds of lines of program code. It's time to take a closer look at this main program file.

Your programs have in fact consisted of a single *function*, called `main`. The start of this function marks the start of the program:

```
int main ()
```

and the set of statements forming the program have made up the *body* of this function. A single pair of braces has been used to mark the beginning and end of this set of statements.

You may well have noticed that this first line of the main function looks very much like the definition (or *prototype*) of a method in the header file of a class. The format is in fact exactly the same; there is a return type, the name of the function (or method), and finally some brackets that can contain the types of some parameters[1]. There is a good reason for this; methods are functions too. Methods are functions that are attached to a particular class.

This means that you have already met two types of function. Methods are functions that form part of a class, and the `main` function is a special function that forms the body of a program that uses the class.

This chapter describes how to use two further types of function. The first, *non-member functions*, are functions that are very much like methods but which are not attached to any class. The second, called *built-in functions*, are provided by C++ to carry out a whole range of common tasks.

Non-member functions

You need to learn no new C++ in order to use non-member functions. They are specified and used in a way that is very similar to methods. The prototype of one has exactly the same format as the definition of a method:

<return type> <function name> (<parameters>);

1 The main function can, in fact, have parameters. A particular case where this is useful is described in the last chapter of the book.

The only difference is that a non-member function is not associated with any class. It is therefore not defined in a class's definition or implementation.

The definition of a non-member function is split into two in the same way as a method. There is a prototype that specifies how the function works, and a separate implementation. It is possible to use a function based only on information from the prototype; it is not necessary to see the implementation.

Any program that uses a non-member function must include the function's prototype[2]. The prototype is added to the program file, normally after any `#include` lines but before the start of the `main` function:

```
#include <iostream>

using namespace std;

// prototypes of non-member functions go here

int main ()
{
  // main function statements

  return 0;

}
```

The prototypes have exactly the same format as the familiar definition of a method, except that the optional `const` keyword is not applicable since there is no instance of a class to remain constant when the function is called. For example, a non-member function might be used to convert a `char` variable storing a letter in lower case to its upper-case equivalent; its prototype would be:

```
char upperCase (char);
```

This is interpreted in exactly the same way as the prototype of a method. The function is called *`upperCase`*, it returns a `char` value, and it takes a single `char` value as a value parameter.

The implementation of the function can be written in one of two places. It can be included at the bottom of the single program file after the final closing brace:

```
#include <iostream>

using namespace std;

// prototypes of non-member functions go here

int main ()
{
  // main function statements

  return 0;
```

2 Another statement that's not quite true, I confess. It's also possible to include the whole function in the program, before the `int main ()`. In this case the prototype isn't necessary, since the compiler will see the whole function definition before the function is used.

```
}

// function implementation goes here
```

Alternatively, and probably better (since the above approach can still lead to very long files), it can be written in its own file and linked with the `main` function when the whole program is compiled.

Your *Local guide* should tell you how to go about writing programs that are stored in many files. Using separate files to store functions can make editing the program very much easier, since smaller files are generally easier to handle in text editors. This approach also makes it much easier to reuse non-member functions in different programs.

The implementation of the function follows much the same format as the familiar implementation of a method except that there is no class name to include:

```
char upperCase (char c)
{
  // implementation here
}
```

Non-member functions are called in the main function in a slightly different way to methods. But, while the syntax is different, the effect is very much the same. To call the *`upperCase`* function on a `char` variable *`ch`* the statement would be:

```
ch = upperCase (ch);
```

This statement would presumably convert the value of *`ch`* to its upper-case equivalent. The only real difference in the syntax is of course that there is once again no class instance to be named.

Finally, non-member functions may also have reference parameters. These parameters have exactly the same effect, and are used in the same way, as reference parameters are with methods. And as before, a parameter is declared to be a reference parameter with the addition of an `&` symbol:

```
void swap (int&, int&);
```

This function is intended to swap the values of two integer variables, quite a common task. This is its prototype, declaring that it doesn't return a value and that it requires two integer variables as reference parameters; there should really of course also be a comment to explain precisely what it does with these parameters.

This function prototype would be included at the top of the program file, above the `main` function. The implementation would then either be included at the bottom of the same program file after the final closing brace or would be stored in its own file. Wherever it is stored the implementation is:

```
// swap two integer values

void swap (int& a, int& b)
{

  int temp = a;

  a = b;
  b = temp;
```

```
  return;

}
```

This function can then be used in the `main` function. The prototype is included at the top and, in this example, the implementation of the function is included below the `main` function.

```
// swap.cc
//
// Swaps the values of two integer variables.
// Demonstrates declaration and use of
//   non-member functions.
//
// Chapter 16.
//
// Author:    AMJ
// Date:      19th December 2001
// Platform:  g++, Linux

#include <iostream>

using namespace std;

void swap (int&, int&);

int main ()

{
  int x, y;

  cout << "Enter value for x: ";
  cin >> x;

  cout << "Enter value for y: ";
  cin >> y;

  cout << "Swapping . . ." << endl;

  swap (x, y);

  cout << "x is now " << x << " and y is now "
       << y << endl;

  return 0;
}

// swap two integer values

void swap (int& a, int& b)
{
  int temp = a;

  a = b;
  b = temp;

  return;
}
```

The *swap* function also declares its own variable, *temp*. This variable can only be used inside this function; it is called a *local variable*. When the function has finished and control has been returned to the `main` function any local variables in the function are deleted and their values are lost forever. This is exactly the same as the use of local variables in methods.

Non-member functions can be used for a variety of purposes. They can be used to reduce the complexity of the `main` function and to share out the development work between many programmers. Functions should definitely be used if any task is to be used more than once in a particular program. Any code that is duplicated in a program should always be written as a function; it can be a huge waste of effort to have to test and correct the same code in a number of different places in a single program.

One of the advantages of using classes in programs is that they allow for code reuse. The same advantage applies to the use of functions; functions that have already been written and tested can be used again and again in other programs in the future.

Many new programmers are tempted to ignore the possibility of using functions at first. They sometimes prefer to continue to write their programs in single long files and to grapple with the unwieldy files that often result from this approach. If they are told that they have to use functions they will often be tempted to write the program in a single file first and to try and split it into functions later. This approach is an extremely complex task and is usually a sure recipe for disaster and a lot of extra work. Don't be tempted to do it!

When a program is designed it should also be split down into functions; this should happen before any of the program is written. The functions should be developed separately in much the same way as a class's methods. After the functions have been written and tested the development of the `main` function is usually little more than the relatively straightforward process of combining all the existing tested and working pieces of program. The use of functions makes the development of programs much easier. Each programmer has only to work on a small part of a program at any one time, and small programs are much easier to write than big ones.

Built-in functions

There are many tasks that are so common and are needed in so many different applications that C++ provides built-in functions to perform them. There are two types that you will probably need to use at some point:

- methods to operate on strings
- functions to operate on built-in data types except strings

These two are separated because behind the scenes strings are actually classes, while the other C++ data types are not. This slight complication is a somewhat unfortunate consequence of the development of C++ from C; `float`, `int` and `char` are C data types, while `string` was introduced as a class in C++[3]. This also means that the format for using these two types of built-in

3 In case you're wondering, there was a different data type for strings in C. It went by the name of `char*`, and is still available in C++. You'll meet it when you come to use some advanced features of C++.

function is slightly different, in much the same way as there are different formats for using methods and non-member functions. This is slightly untidy, but makes perfect sense when you get used to it.

String functions (methods)

There are many functions (or more correctly many methods) for carrying out operations with strings. Many programs need to carry out basic string-processing tasks, so it makes a great deal of sense for a library of common string operations to be provided. The ones that are most common and useful are:

`int length ();`	returns the length of the string
`string substr (m, n);`	returns the substring consisting of *n* characters starting at the *m-1*th character
`char at (n);`	returns the character at position *n-1* in the string
`void append (s);`	appends the string *s* to the string
`int find (s);`	finds and returns the position of the string *s* in the string

Since these functions are all methods they operate on an instance of the `string` class, which is what a string variable actually represents; a string variable is in fact an object. The syntax for calling one of these methods is exactly the same as that for calling any other method of any other class:

<class instance>.<method> (<parameters>);

For example, to use the `length` method to print out the length of a string:

```
string s = "grace hopper";

cout << "The string is " << s.length ()
     << " characters long" << endl;
```

or similarly to store the return value in a variable:

```
string s = "grace hopper";
int length = s.length ();
```

To make use of some of these functions you need to know a little bit about how strings are stored. A string is stored as a sequence of characters, and C++ maintains an index of each character within the string. Unsurprisingly, C++ starts the index at 0; this means that the first character of the string is at index 0. This also means that the last character in a string is stored at the index one less than the length of the string:

```
string s = "ada lovelace";

cout << "The last character of the string is at index "
     << s.length () - 1 << endl
     << "It is " << s.at (s.length () - 1) << endl;
```

It might be a good idea to type these simple examples into a program so that you can see exactly what's going on. Use the programs to experiment with the other string methods.

The `substr` method extracts substrings from other strings and is an example of where an understanding of how the index values work becomes very important. The first parameter of this method is the index at which the

required substring starts and the second is the length of the required substring. Since the index starts at 0 the correct values for the parameters may not be quite as expected, and careless mistakes can be made. For example:

```
string s = "abcdefghijklmnopqrstuvwxyz";
string t;

t = s.substr (0, 3);  // t is "abc"
t = s.substr (1, 3);  // t is "bcd"
t = s.substr (23, 2); // t is "xy"
```

Again, it would be a good idea to try this in a simple program. Perhaps you could write a small program to prompt for two values and extract any substring. Don't forget some error checking! What happens if you try to access characters with an index value that is too big or too small?

Other functions

C++ provides access to a range of functions that operate on the other basic data types. These functions are actually part of the C language, so most of them require that particular C libraries are included. There is no need to include prototypes of these functions, since these can be found in the system libraries and will be found automatically by the compiler[4]. Your *Local guide* or the documentation of your C++ system should provide you with accurate and precise prototypes and descriptions for these functions.

The first group of built-in functions is those that can be used with the numeric data types to calculate various useful mathematical values:

`float sqrt (float);`	square root
`float sin (float);`	sine
`float cos (float);`	cosine[5]
`float atan (float);`	inverse tangent
`int ceil (float);`	rounds down
`int floor (float);`	rounds up
`int abs (int);`	absolute value (removes the sign)
`float fabs (float);`	as `abs ()`, but for floating-point values
`float pow (float, float);`	calculates powers

These functions are all called in exactly the same way as non-member functions. They all require that the C maths library is included. The `sqrt` function (which we already used in an example in the previous chapter) for example is used with the C maths library `cmath`:

```
// sqrt.cc
//
// Calculates the square root of a number using the
```

4 This is, in fact, exactly what the `#include` lines in your programs are doing. They are including prototypes for the built-in functions in C++.

5 There is, strictly speaking, no function for tangent, since it can be calculated by dividing the sine by the cosine. This is not especially neat, and you will probably find that your C++ system also provides a `tan` function, together with many more.

```
//   sqrt function from the C maths library.
//
// Chapter 16.
//
// Author:    AMJ
// Date:      19th December 2001
// Platform:  g++, Linux

#include <cmath> // for sqrt
#include <iostream>

using namespace std;

int main ()
{
  float x;

  cout << "Enter a number: ";
  cin >> x;

  cout << "The square root of " << x << " is "
       << sqrt (x) << endl;

  return 0;
}
```

Again your *Local guide* or other documentation should tell you precisely which libraries are required for which functions. These libraries are simply included together with any others at the start of your program. It's possible that your system may list C libraries in the form:

```
#include <ctype.h>
```

with a *.h* extension in the same way as a header file. The correct way to include these libraries in a C++ program (rather than a C program) is to remove the *.h* and to add a lower-case *c* to the start of the library name to show that the library is part of C. So to include the C *ctype* library in a C++ program the include line is:

```
#include <cctype>
```

The *Local guide* should show you for certain how to include these libraries.

A common task in many programs is to process a character variable and to take some action depending on what type of character it is. This approach is often used to validate that some data entered by a user is as required, for example. Readily identifiable types of character include spaces, alphabetic, alphanumeric and digits:

`bool isspace (char);`	`true` if the character is a space, `false` otherwise
`bool isalpha (char);`	`true` if the character is alphabetic, `false` otherwise
`bool isalnum (char);`	`true` if the character is alphanumeric, `false` otherwise
`bool isdigit (char);`	`true` if the character is a digit, `false` otherwise

These functions also require the inclusion of the *cctype* library.

Two more functions also in *cctype* allow the detection of specific types of alphabetic character:

`bool isupper (char);`	`true` if the character is an upper case letter, `false` otherwise
`bool islower (char);`	`true` if the character is a lower case letter, `false` otherwise

These functions are commonly used in Boolean expressions and conditions:

```
// type.cc
//
// Determines the type of a character using functions
//   from the C stdlib library.
//
// Chapter 16.
//
// Author:    AMJ
// Date:      19th December 2001
// Platform:  g++, Linux

#include <cctype>
#include <iostream>

using namespace std;

int main ()
{
  char c;

  cout << "Enter a character: ";
  cin >> c;

  if (isspace (c)) {
    cout << "a space" << endl;
  }
  else if (isdigit (c)) {
    cout << "a digit" << endl;
  }
  else if (isalpha (c)) {
    cout << "alphabetic" << endl;
  }
  else {
    cout << "something else!" << endl;
  }

  return 0;
}
```

As well as detecting whether an alphabetic character is an upper- or lower-case letter it is also possible to convert the character between the cases:

`char toupper (char);`	returns upper case equivalent of character if appropriate
`char tolower (char);`	returns lower case equivalent of character if appropriate

In both cases, if it is not sensible to convert the character (for instance if it is a digit or punctuation symbol or if it is already in the required case) it is left unchanged. For example:

```
// case.cc
//
// Demonstrates the built-in functions for changing the
```

```
//   case of a character.
//
// Chapter 16.
//
// Author:    AMJ
// Date:      19th December 2001
// Platform:  g++, Linux

#include <cctype>
#include <iostream>

using namespace std;

int main ()
{
  char ch = 'a';

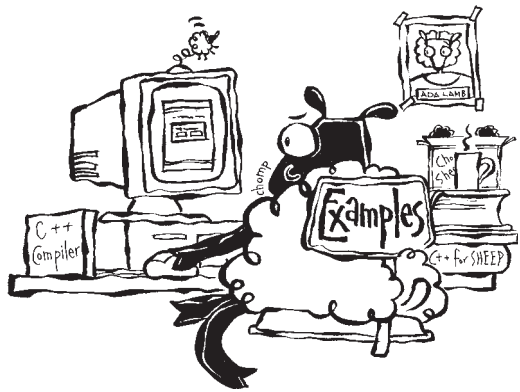
  cout << ch << endl; // outputs 'a'
  ch = toupper (ch);  // converts ch to upper case
  cout << ch << endl; // outputs 'A'
  ch = toupper (ch);  // no effect
  cout << ch << endl; // outputs 'A'

  ch = '!';
  ch = tolower (ch);  // has no effect
  cout << ch << endl; // outputs '!'

  return 0;
}
```

There are many other built-in functions available, but these are probably the most useful. Some C++ systems will provide more functions than others and some will work slightly differently on different systems. Your *Local guide* or other reference should provide you with a full list of the prototypes of all the functions available on your system.

There has been very little new C++ in this chapter, but there has been a vitally important new idea. The key to writing extensive useful programs is the ability to break the program down into smaller parts that can be shared out among a team of programmers. This means breaking down the program into functions, and it is crucial that this is done before the program is started rather than after. It's time to get some practice in writing some functions.



Example 1 – A name is a name

Mollie is writing a program to store details of all the sheep. The program stores a great deal of information about each sheep, all of which is entered by Farmer Strawstrap. Unfortunately, the farmer often makes mistakes when entering this data, so Mollie has decided that the program needs a set of functions to validate the data that is entered.

A specific problem is that the farmer is inconsistent when entering the names of the sheep. Mollie wants to make sure that all the names are made up of alphabetic characters only and that the first letter is in upper case with the others in lower case. As part of dividing up the work of developing the program Mollie has asked Flossy to develop a function to do this validation. Flossy has been told to assume that all the sheep have exactly one name.

This function will take a string as a parameter and return a Boolean value so its prototype is:

```
bool validName (string);
```

The function is going to need to test each character in the name in turn. The built-in functions `isupper` and `islower` will provide the necessary ability to check each character, and the `at` method of the `string` class will allow each character to be extracted in turn. For example, this test will examine the first character of the string *name* and check whether it is an upper-case character:

```
if (isupper (name.at (0)))
```

The method call `at (0)` extracts the first character and this is then tested by the `isupper` function.

After this test the remaining characters can be examined in turn. This is an obvious application for a loop, specifically a determinate loop, as the number of characters to be tested is available from the `length` method.

```
for (int i = 1; i < n.length (); i ++)
```

This loop starts at 1, which is the position of the second character of the string. It will allow each character in the remainder of the string to be extracted in turn with the `at` method and then tested with the `islower` built-in function.

```
for (int i = 1; i < n.length (); i ++) {
  if (!islower (n.at (i))) {
    // invalid character found!
```

```
    }
  }
```

If either of these two tests finds an invalid character then the function should return `false`. Otherwise (if the loop finishes) it should return `true`.

Finally the function must be tested. It cannot be compiled on its own, so a simple driver program is required so that the function can be compiled and run. It is quite usual to write small driver programs so that functions can be tested before they are used in other programs.

```
// validName.cc
//
// Tests validName function to validate a name.
//
// Chapter 16, Examples 1.
//
// Author:    AMJ
// Date:      18th December 2001
// Platform:  g++, Linux

#include <cctype> // for isupper and islower
#include <iostream>
#include <string>

using namespace std;

bool validName (string);

int main ()
{
  string name;

  cout << "Enter the Name: ";
  cin >> name;

  if (validName (name)) {
    cout << "\"" << name << "\" is valid" << endl;
  }
  else {
    cout << "\"" << name << "\" is not valid" << endl;
  }

  return 0;

}

bool validName (string n)
{

  if (!isupper (n.at (0))) {
    return false;
  }
  else {
    for (int i = 1; i < n.length (); i ++) {
      if (!islower (n.at (i))) {
        return false;
      }
    }
  }
```

```
    return true;

  }
```

Example 2 – Users and email

Mollie has another problem with her data. All the sheep have been given a unique username for their computer system, but her files store only the sheep's email address. Happily the email address contains the username, as it is in the form "`username@strawstrap.com`" with the username being formed from all the characters before the `@` symbol. Mollie wants a function that will extract the username from the email address.

This function is going to return a string (the username) and requires a string (the email address) as a parameter. Its prototype is:

```
string findUser (string);
```

Mollie needs a function that will find the `@` symbol in a string. This is simply done by traversing the string from left to right examining each character in turn. A counter is kept of the position being examined and, when the `@` symbol has been found, this variable will store the correct place in the string. Since the symbol can be anywhere in the string the loop is indeterminate; there is no point in searching through the whole string if the `@` is found near the start. A neat solution is to use a Boolean variable *`found`* to indicate whether the symbol has been found.

Assuming that the email address is stored in a string variable *`email`* the loop is:

```
int atPosn = 0; // position of @ symbol
bool found = false;

while (atPosn < email.length () && !found) {
  // check character
  atPosn ++;
}
```

The check on the character is a case of using the `at` method of the `string` class:

```
if (email.at (atPosn) == '@') {
  found = true;
}
```

To make sure that the correct position is stored when the character has been found *`atPosn`* is incremented only if the `@` has not been found:

```
if (email.at (atPosn) == '@') {
  found = true;
}
else {
  atPosn ++;
}
```

This can be added to the loop:

```
int atPosn = 0; // position of @ symbol
bool found = false;

while (atPosn < email.length () && !found) {
```

```
    if (email.at (atPosn) == '@') {
      found = true;
    }
    else {
      atPosn ++;
    }
  }
```

Once the `@` has been located, the username can be extracted from the string using the `substr` method of the `string` class. This method takes two arguments: the starting position of the substring and the required number of characters. In this case the starting position is always the first element of the string, at position 0. The number of characters required is the index of the `@` symbol. The statement to extract the substring and return it from the function is:

```
return email.substr (0, atPosn);
```

A final possibility is that there is some error in the email address and there is no `@` to be found. In this case the function must return some suitable value to indicate that this error has happened; an empty string would seem a good choice. This situation will be easily detected since *found* will be false:

```
if (found) {
  return email.substr (0, atPosn);
}
else {
  return "";
}
```

Any program using this function will have to detect and handle this error case.

As usual a small driver program is needed to test the function. One possibility that includes a complete version of the function is:

```
// username.cc
//
// Extracts a username from an email address.
//
// Chapter 16, Examples 2.
//
// Author:    AMJ
// Date:      18th December 2001
// Platform:  g++, Linux

#include <iostream>
#include <string>

using namespace std;

string findUser (string);

int main ()
{
  string email;
  string username;

  cout << "Enter the Email Address: ";
  cin >> email;
```

```
    username = findUser (email);

    cout << "Username is \"" << username << "\"" << endl;

    return 0;

  }

  string findUser (string email)
  {
    int atPosn = 0;
    bool found = false;

    // find @ character

    while (atPosn < email.length () && !found) {
      if (email.at (atPosn) == '@') {
        found = true;
      }
      else {
        atPosn ++;
      }
    }

    // return substring if found, or blank otherwise

    if (found) {
      return email.substr (0, atPosn);
    }
    else {
      return "";
    }
  }
```

It is interesting that, while the function is quite complex, the program that uses it is very simple. This is a fine example of one of the advantages of using functions to split a program into small, more manageable, parts.

An alternative approach to developing this function involves developing a rather more general solution by separating out the task of finding a given character in a string. It makes a lot of sense to do this, since this is such a common task that the function to do it may be of future use, or there may already be a version available.

A function to find a character in a string and return its location would have the prototype:

```
int locate (string, char);
```

The code is essentially the first part of the *findUser* function in the previous program. At the same time the *findUser* function itself is shortened to make use of the new function. The full program rewritten in this way is:

```
// username.cc
//
// Extracts a username from an email address.
// Version using a locate function.
//
// Chapter 16, Examples 2.
//
```

```
// Author:    AMJ
// Date:      18th December 2001
// Platform:  g++, Linux

#include <iostream>
#include <string>

using namespace std;

string findUser (string);
int locate (string, char);

int main ()
{
  string email;
  string username;

  cout << "Enter the Email Address: ";
  cin >> email;

  username = findUser (email);

  cout << "Username is \"" << username << "\"" << endl;

  return 0;

}

string findUser (string email)
{
  int atPosn = locate (email, '@');

  if (atPosn != -1) {
    return email.substr (0, atPosn);
  }
  else {
    return "";
  }
}

int locate (string s, char c)
{
  int atPosn = 0;
  bool found = false;

  while (atPosn < s.length () && !found) {
    if (s.at (atPosn) == c) {
      found = true;
    }
    else {
      atPosn ++;
    }
  }

  if (found) {
    return atPosn;
  }
  else {
    return -1;
```

```
    }
}
```

The *locate* function also has to deal with the possibility that the character is not found in the string; since it returns an integer, the simplest way to do this is to return –1 (a value that is clearly not an index).

Example 3 – Passwords

Mollie is concerned that some of the sheep are using poor passwords on the farm's computer system. The password "baa" is worryingly popular and "grass" is little better. Mollie would like a function that will test a password as a sheep chooses it and will report how good the password is.

Mollie has devised a simple scale to assess passwords. She has decided that all passwords should include upper-case letters, lower-case letters, digits and punctuation (defined as any other character). One point is allocated to a password when each type of character is found, to a maximum of four. For example "baa" contains only lower-case letters and would score one point. "Baa" is slightly better and would score two. Better still is "Baa!" which would score three. The ideal password would naturally be "8aa!"[6].

This function is simply going to examine each character of the password in turn. It will return the point score, which is an integer:

```
int checkPassword (string);
```

The built-in functions `isupper`, `islower`, and `isdigit` will serve to determine the type of each character and a conditional statement will trap the possibility of any other character and assume that it is punctuation. If the character being examined were stored in *ch* the conditional statement would be:

```
if (isupper (ch)) {
  // upper case letter found
}
else if (islower (ch)) {
  // lower case letter found
}
else if (isdigit (ch)) {
  // digit found
}
else {
  // assume punctuation found
}
```

This statement is simply enclosed in the usual determinate loop to examine each character in a string in turn:

```
for (int i = 0; i < s.length (); i ++)
```

The complete function works by declaring four integer variables, one for each type of character. These are initially set to 0 and are set to one only when a suitable character is found. The final value returned from the function is the sum of these found variables.

The complete function is included in this program:

---

6 These are all, of course, still very poor passwords. Do not be tempted to use them.

```
// password.cc
//
// Tests function to check use of various characters
//   in a password.
//
// Chapter 16, Examples 3.
//
// Author:    AMJ
// Date:      18th December 2001
// Platform:  g++, Linux

#include <cctype>
#include <iostream>
#include <string>

using namespace std;

int checkPassword (string);

int main ()
{
  string password;
  int score;

  cout << "Enter the Password: ";
  cin >> password;

  score = checkPassword (password);

  switch (score) {
    case 4: cout << "A fine password" << endl;
            break;
    case 3: cout << "A poor password" << endl;
            break;
    case 2: cout << "A very bad password" << endl;
            break;
    case 1: cout << "An abysmal password" << endl;
            break;
    default: cout << "Error. Empty password?" << endl;
             break;
  }

  return 0;

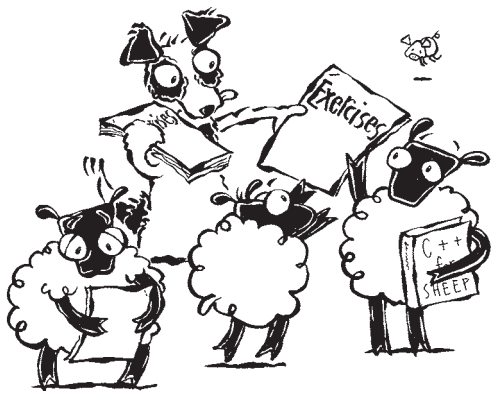
}

int checkPassword (string p)
{
  int hasUpper = 0;
  int hasLower = 0;
  int hasDigit = 0;
  int hasPunc = 0;

  for (int i = 0; i < p.length (); i ++) {
    if (isupper (p.at (i))) {
      hasUpper = 1;
    }
    else if (islower (p.at (i))) {
      hasLower = 1;
```

```
    }
    else if (isdigit (p.at (i))) {
      hasDigit = 1;
    }
    else {
      hasPunc = 1;
    }
  }

  return hasUpper + hasLower + hasDigit + hasPunc;
}
```



16.1 Write a program that prompts the user to enter three integers and then displays the average. Beware of integer division! If the average is not a whole number the program should not display the decimal part. For example, 2.5 should be displayed as 2, as should 2.95 and 2.15. A hint would be to look back through the list of built-in functions available for numeric types.

16.2 Modify your program to display the average rounded to the *nearest* integer. If two integers are equally nearest the higher of the two should be displayed.

16.3 Write a function *`isWord`* that takes a string and returns `true` if the string is made up of alphabetic characters only and `false` otherwise. Use a small driver program to test the function.

16.4 A class to store dates has a method *`setDate`* that takes a string in the form *`DD/MM/YYYY`* and converts it to three separate private string attributes. The header file is:

```
using namespace std;
class Date {
  private:
    string day, month, year;
  public:
    Date ();
    bool setDate (string);
};
```

Implement this method, and test it by writing a short program and adding a suitable method to output the date. If the string provided to the method is

not in the expected format, all three of the private attributes should be set to empty strings. The method should return a Boolean value which should be `true` if the string was in the correct format and `false` otherwise. You will need to add another method to print the date so that you can see what's going on!

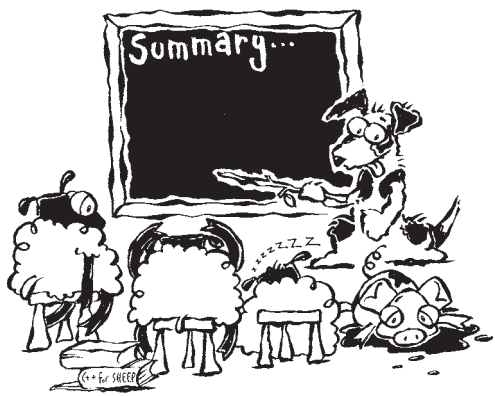
16.5 Write a program that analyses a sequence of words input by the user. It should find the number of words, the number of characters (excluding punctuation and spaces) and the number of sentences.

Assume that:

- all sentences are terminated by a full stop, exclamation mark or question mark
- all words are separated by spaces, or are at the end of a sentence
- the input is ended with the word END in capitals
- the word END does not appear elsewhere

The final END should not be included in the analysis.

16.6 Extend your program to display the longest and shortest words input and the average length of all the words. If there is more than one word of the longest or shortest length the first found in the text should be displayed.



A program of any complexity is going to require several hundred lines of code. Handling such a program in a single file is very cumbersome, so programs should be broken down into much smaller units, called functions. Functions are very much like the methods of a class except that they are not members of any class; they are often called non-member functions. A complete program will make use of many functions; these may well be the work of many different programmers. As well as splitting up a complex task the use of functions also means that a library of useful functions can be built up to allow for code reuse.

The specification of a non-member function is very like the specification of a method in a class. The prototype gives the name of the function, the type of value that it returns and the types of any parameters that it requires. Prototypes of non-member functions must be included at the top of any programs that use them. The implementations can also be included in the same file, at

the bottom, but it is usually better to write them in a separate file. Non-member functions cannot be compiled and run on their own, so it is often necessary to write small programs (called *drivers*) so that they can be tested.

Since some tasks are very common there are a number of built-in functions provided in C++. Many of these are inherited from the C language and so require that various C libraries are included in the C++ programs that use them. Your *Local guide* should provide you with a list of the functions available on your C++ system and should tell you which libraries must be included. It is always a good idea to consult the list of provided functions before starting to write your own; there is no point in writing a function that is already provided!

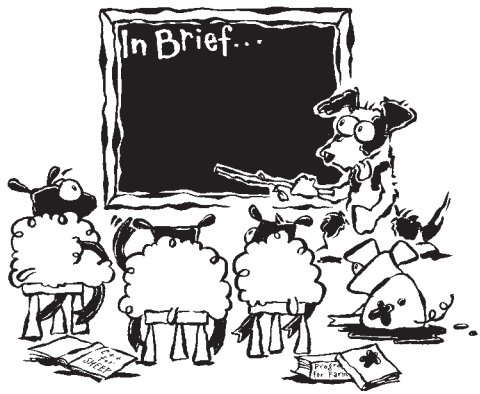
You should now be able to write complex programs that make use of functions. You should be able to identify parts of a program that might be best written as functions and you should be able to write short programs to test your functions. More importantly you should understand how to make use of the built-in functions and also functions that have been written by other programmers.

You have now in fact seen almost all the C++ that you will find in this book. But there is one last thing; sometimes variables are needed that can hold more than one value...



Chapter 17
Collections of Things





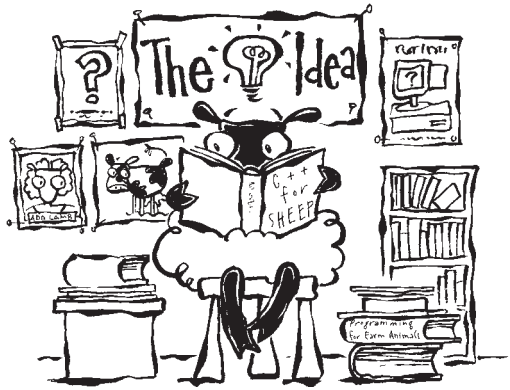
Variables can store only one value at any one point in time.

While this may seem a very obvious observation, it represents an important limitation of the programs that you have written so far. Programs are often required to process collections of related data; examples might be a set of grades, a flock of sheep or a pile of books. It is possible to handle collections of data by declaring and using corresponding collections of variables, but this is often very cumbersome. A neater mechanism is needed.

This chapter explains how collections of data can be handled in C++ programs. There are two similar structures available that can be used to store collections of values. The structure for doing this in C, which is also of course available in C++, is called an *array*. The equivalent technique in C++ is called a *vector*. Vectors are more flexible than arrays in several ways, but in most applications the two are just about equivalent[1]. Both can be used to implement objects that are effectively collections of other objects or values.

After reading this chapter you should understand how programs can be written that handle objects that are themselves collections of other objects. You should understand how this can be done in C++ with arrays and vectors, and you should understand the similarities and differences between these two structures. You should also understand a little more about the way in which string variables are stored.

1 An important reason why you should know about arrays is that arrays are found in almost all programming languages, while structures such as vectors are found in fewer languages.



Before starting to use arrays and vectors it will help if you understand some final details about the way in which strings are stored in C++. This will help because strings are basically a special case of an array or vector; they are an array or vector of characters that usually happens to spell out some useful text when arranged in sequence. You should be quite familiar with using strings by now, so a proper understanding of how they work behind the scenes will help you understand more general arrays and vectors.

More about strings

In C++ strings are stored as sequences of characters. Some of the programs in the previous chapter used the `at` method of the string class to extract individual characters from strings. Strings are, in fact, stored in a very similar way to arrays and vectors. Each character in a string is said to be an *element* of the string. Each element in the string has a unique *index* value, assigned sequentially from the first element of the string (which is stored at index number 0). These are the index values used by the `at` method.

Any particular character in a string can be referenced by its index number without using the `at` method; the syntax is to enclose the index number in square brackets after the name of the string:

```
string str;

cout << str[6];   // character at index number 6
cout << str[0];   // first character
```

The first of these is equivalent to:

```
cout << str.at (6);
```

As an example, the first character of a string can be extracted and displayed:

```
string str = "maurice wilkes";

cout << "The first character of the string is "
     << str[0] << endl;
```

The same applies to any other character in the string.

It is important to be careful not to try and reference a character using an index value that is too big:

```
string str = "ada lovelace";

cout << str[0];   // fine, outputs 'a'
cout << str[20];  // Error! The index is too big
```

The error here is that the program is attempting to access part of the computer's memory beyond the string. Errors such as this will probably not be spotted by the compiler and will instead cause the program to fail while it is running. This failure might cause the program to stop with some internal error message or, worse, the program might continue to execute, producing incorrect results. It all depends on what is being stored at the memory that is being accessed by mistake.

It is therefore very important to know when a string has finished, or put another way, the index value of the last character. The last character can in fact always be referenced by making use of the built-in method to determine the length of a string. Since the first character is at index 0 the index of the last character is one less than the length of the string:

```
string str = "charles babbage";

cout << "The last character of the string is "
     << str[str.length () - 1] << endl;
```

There are built-in functions to carry out some conversions of `char` variables; an example is `toupper ()`, which will convert a letter held in a `char` variable to its upper-case equivalent. While there is no built-in version of this function to convert entire strings, it can be implemented easily enough by using a `for` loop to convert each character in turn:

```
string str = "linus torvalds";

for (int i = 0; i < str.length (); i ++) {
  str[i] = toupper (str[i]);
}
```

The counter variable of the `for` loop causes the operation to be carried out for each character in turn, one each time the loop is executed. This loop is equivalent to the sequence of statements:

```
str[0] = toupper (str[0]);
str[1] = toupper (str[1]);
str[2] = toupper (str[2]);
...
str[str.length () - 1] = toupper (str[str.length () - 1]);
```

The crucial advantage of using a loop is that the program will work correctly no matter how long the string turns out to be. If the operation were to be coded in individual statements the programmer would always have to know in advance how long the string was going to be.

This is an extremely common form of operation on strings and indeed also on arrays and vectors. A `for` loop is used to carry out some operation on each element of the collection in turn. Another example is a non-member function to determine whether a string contains a space:

```
bool includesSpace (string s)
{
  for (int i = 0; i < s.length (); i ++) {
    if (isspace (s[i])) {
```

```
        return true;
      } // if
    } // for

    return false;
  }
```

or another to return the index of the first location of a certain character in a string:

```
int locateChar (char c, string s)
{
  for (int i = 0; i < s.length (); i ++) {
    if (s[i] == c) {
      return i;
    } // if
  } // for

  return -1; // indicates not found
}
```

Strings, then, are simply a special case of a variable that can store a collection of values; they store a collection of characters (type `char`). They are a built-in type because strings are a very common type of data to process in programs. Arrays and vectors allow collections of other data types to be stored and used.

Arrays

An array is a collection of values of a particular type. An array can contain collections of values of any type, such as integers, floating-point numbers and objects. The declaration of an array specifies the name of the array, the type that can be stored and the number of items in the collection:

<type> <array name>[<number of items];

For example:

```
int numbers[10];    // array of 10 integers
float averages[6];  // array of 6 floating-point numbers
string names[11];   // array of 11 strings
```

An array can store values of only one type – the type that is specified in the declaration.

Each value in the array is called an *element*. As with a string, each element is stored at an *index*, with the first element at index 0. The declaration can also provide initial values for the elements:

```
int numbers[4] = {1, 2, 3, 4};
string days[7] = {"Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat"};
```

The number of elements in the array can also be taken from the value of an integer constant:

```
const int NumberOfValues = 10;

int numbers[NumberOfValues]; // array of 10 integers
```

The elements of an array declared in this way can also be given initial values.

```
int numbers[NumberOfValues] = {10, 9, 8, 7, 6, 5, 4, 3, 2, 1};
```

Declaring the number of elements in an array as a constant is quite common. It allows for the possibility that the number of values to be stored in an array will change and makes the program easier to modify since only one value need be altered. A useful side-effect is that the name of the constant can provide some useful information about the size of the array:

```
const int NumberOfPlayers = 4;

int scores[NumberOfPlayers] = {0, 0, 0, 0};
```

An individual element of an array can be referenced using the familiar syntax for referring to a single character in a string:

```
int numbers[4] = {1, 2, 3, 4};

cout << numbers[1]; // 2
cout << numbers[0]; // 1
```

The elements can also be used in expressions and in any way that a variable of the same type can be used:

```
int numbers[4] = {1, 2, 3, 4};

cout << numbers[0] + numbers[1] + numbers[2]; // 6
```

If initial values are not provided when the array is declared, a common task is to set each element of an array to some initial value. This is usually achieved with a `for` loop:

```
int scores[10];

for (int i = 0; i < 10; i ++) {
  scores[i] = 0;
}
```

A potential source of errors in this example is that the number of elements in the array (here 10) has to be included twice, once in the declaration of the array and then again in the control statement of the loop. There is no equivalent to the `length` method of the `string` class that can be used with arrays. This is where the use of a constant can help:

```
const int NumberOfScores = 10;
int scores[NumberOfScores];

for (int i = 0; i < NumberOfScores; i ++) {
  scores[i] = 0;
}
```

This means that the number of elements in the array need be included in the program only once. If the value of *`NumberOfScores`* had to be changed there would be only one change needed in the program, and this would change the size of the array as well as the workings of the loop.

Arrays can be used as parameters to methods and functions. In most cases it is necessary to pass two values to the function. The first is the array itself and the second is the number of elements in the array. The number of elements in the array is omitted from the array in the function prototype, but the square brackets are retained to show that the parameter is an array. This allows functions to be written that can deal with arrays of any size. The number of elements in the array is passed as a separate parameter.

For example, a function to find the mean value of an array of integers would have the prototype:

```
float mean (int[], int);
```

to indicate that the first parameter is an array of integers and the second is a single integer (the number of elements in the array). The implementation of this function involves the usual `for` loop to process each element of the array to build up a total, followed by some division. The number of elements is provided to the function and is used to control the loop. The full implementation is:

```
float mean (int numbers[], int elements)
{

  int total = 0;
  for (int i = 0; i < elements; i ++) {
    total += numbers[i];
  }

  return float (total) / elements;
}
```

This would be called from a program simply by specifying the name of the array as a parameter. Again, the use of a constant makes the program easier to follow and maintain:

```
const int NumberOfScores = 6;

int scores[NumberOfScores] = {12, 23, 24, 11, 23, 45};

cout << "The average score is "
     << mean (scores, NumberOfScores) << endl;
```

As another example, a function to find the highest element in an array of integers:

```
int highestElement (int numbers[], int elements)
{
  // Assume first element is highest

  int highest = numbers[0];

  // Loop through the other elements

  for (int i = 1; i < elements; i ++) {
    if (numbers[i] > highest) {
      highest = numbers[i];
    }
  }

  return highest;
}
```

The prototype for a function that uses an array simply declares the type as usual, using empty square brackets to show that the parameter is an array:

```
float mean (int[], int);
int highestElement (int[], int);
```

The number of elements is never specified in the prototype.

It is significant that both these example functions use a `for` loop to examine each element of an array in turn. This is by far the most common operation that is carried out on arrays.

A complication is that arrays are always passed as reference parameters. This means that the values held in an array can always be changed by a function that uses one as a parameter. To stop this happening an array can be declared as a *constant parameter*, and this is always good practice if it is important that the values in the array are preserved.

Better prototypes for the two example functions would be:

```
float mean (const int[], int);
int highestElement (const int[], int);
```

Any attempt to alter the values in the array in either of these functions will result in an error.

Arrays are part of the C language and are sometimes a little cumbersome to use. The obvious case of this that has cropped up in these examples is the necessity to pass the number of elements in an array together with the array itself. There is no way to determine the number of elements in an array; the value must be stored and maintained in some other variable[2]. To address this C++ provides a slightly different implementation of the same idea – the *vector*.

Vectors

Like a string, a vector is defined as a class[3]. This means that there are a variety of methods available to perform various useful operations on vectors, conveniently including one to find the number of elements. Vectors are extremely powerful and provide features that are way beyond the scope of this book. This section gives just a taster and shows how vectors can be used to achieve the same results as arrays, and how they can often be very much neater.

The key difference between vectors and arrays is that vectors are dynamic. The number of elements that an array can store is specified when the array is declared and can never be changed. Vectors, on the other hand, have no such limit and can grow[4] as required. Like an array, a vector can store values of one type. The syntax for declaring a vector looks slightly odd at first[5]; the word `vector` is followed by the name of the type stored in the vector:

vector <<vector type>> <vector name>;

The name of the type to be stored in the vector is enclosed with angled brackets, < and >. For example:

```
vector <int> numbers;   // a vector of integers
vector <string> names;  // a vector of strings
```

If a program is going to make use of vectors it is also necessary to include the library containing the various methods for the vector class:

```
#include <vector>
```

---

2 And it is probably obvious that things would get terribly confusing if the variable for the number of elements ended up storing an incorrect value!

3 Oh, all right. A vector is actually a template. Don't worry about it.

4 Or, indeed, shrink (but not automatically).

5 This is not helped by the convention of using < and > in these examples. This is the only place where the convention breaks down. Sorry!

Optionally the declaration may also include the initial number of elements in the vector:

```
vector <int> numbers (10);  // a vector of 10 integers
```

This is not essential, but it can be included if the number is known in advance. The number is not fixed.

Values are added to a vector using the `push_back` method. This adds the value by creating a new element at the end of the vector; there is no limit on the number of times it can be used and therefore no limit to the size of a vector.

For example this declaration creates an empty vector:

```
vector <int> numbers;   // vector is empty
```

and this line adds an element to the vector and stores the value 10 there.

```
numbers.push_back (10);
```

Elements in a vector are indexed in exactly the same way as elements in an array. This vector now has one element, stored at index 0. Individual elements of a vector can be referenced in exactly the same way as elements of an array, so:

```
cout << numbers[0];
```

would display this first value. Obviously great care must be taken not to attempt to access index values beyond the current vector.

To make this easier the `size` method of the `vector` class returns the number of elements currently in the vector. This can be used in a loop to display all the elements of a vector:

```
for (int i = 0; i < numbers.size (); i ++) {
  cout << numbers[i] << endl;
}
```

or in a check to ensure that a requested index value is within the vector:

```
int index;

cout << "Enter Index to Display: ";
cin >> index;

if (index < numbers.size ()) {
  cout << numbers[index] << endl;
}
else {
  cout << "No value at that index!" << endl;
}
```

Vectors can also be used as parameters to functions and methods but the existence of the `size` method means that there is no need to pass the number of elements in the vector as an additional parameter. This somewhat simplifies the function to find the mean value of a vector:

```
float mean (vector <int> numbers)
{
  int total = 0;

  for (int i = 0; i < numbers.size (); i ++) {
    total += numbers[i];
```

```
    }

    return float (total) / numbers.size ();
}
```

The prototype of such a function simply specifies the type of the vector:

```
float mean (vector <int>);
```

There are many more methods that can be used with vectors, but the few described here will do for now. These should be enough to show you the difference between arrays and vectors and to allow you to choose which one to use for a particular task.

The use of vectors as parameters in this way raises an issue of program efficiency. This example is fine, and will work, but is far from efficient if the vector contains many hundreds of values; all of these values would have to be copied and the computer would be storing two copies of everything. There is no need to do this, and it is wasteful.

This would not happen if the vector were passed as a reference parameter. There is no need to copy anything, and the extra memory use would be minimal. The function would be declared:

```
float mean (vector <int>&);
```

This is attractive, but would raise the possibility that the function would alter the vector, perhaps in error. The way round this is to declare the parameter as a constant reference:

```
float mean (const vector <int>&);
```

This will be far more efficient. This approach is only really necessary when very large objects are being passed, but it can greatly improve efficiency when they are[6].

Arrays or vectors?

Arrays and vectors can be used for the same tasks. C++ has inherited arrays from C while vectors are recent additions in C++; this probably explains why vectors are rather more sophisticated. The two advantages of vectors over arrays described in this chapter are:

- Vectors can grow to any size. There is no need to declare the upper limit on their size when the program is written and there is no chance that the vector will prove to be too small. This is much more flexible than arrays, which have a static size.
- The `size` method allows a program to determine the number of elements in the vector whenever this value is needed. If the same value is needed when an array is being used it has to be stored and maintained in a separate variable. This is much more convenient and less error-prone.

This might well convince you that vectors are to be preferred in most cases. This is probably reasonable, but some experienced programmers will sometimes prefer the more traditional and less complicated array. More to the

6 Much the same applies to any parameter that uses a lot of memory; this technique can also be used with parameters that are strings or any object type.

point, arrays can be found in most if not all current programming languages and vectors are specific to C++ (even if structures that resemble vectors can now be found in an increasing number of languages). Vectors are a fine choice for a C++ program, but it is essential that you understand arrays so that you can use them in other languages.

Sorting

Perhaps the most common operation on a collection of values is to sort them into some sort of useful order. Once the values have been stored in a vector or an array it is reasonably straightforward to write a function that will sort them based on some key value and will return the array or vector in sorted order.

There are many ways of sorting data, called *sorting algorithms*, and whole books have been written about the relative merits of each one. Some are able to sort large amounts of data very quickly, while others are easy to program and understand. For the moment you are going to be writing programs that use relatively small amounts of data and so speed is not especially important.

The sorting algorithm described in almost all introductory programming books is the same, called *Bubblesort*, and it would be such a shame to deviate from another fine tradition that this is the one described here. Bubblesort is not especially efficient, but it is easy to program and understand.

First a description of how Bubblesort works. The basic idea is that each adjacent pair of elements in a collection is compared and is swapped if the values in the pair are found to be in the wrong order. This process is carried out for each pair in the collection until the collection is in order. If you imagine sorting a line of people into order with the tallest on the left and the shortest on the right, you'll get the idea.

It happens that if this whole process is carried out the same number of times as there are elements in the collection the collection will always then be in the required order. This is obvious if you try it out and examine the intermediate stages in the sorting.

For example, suppose this list of integers was to be sorted into ascending order:

```
6 7 4 2 1
```

The first stage in Bubblesort would compare the 6 and 7, decide that they were in the correct order and make no change. The second comparison would be between the 7 and 4, which are in the wrong order and would be swapped over to give:

```
6 4 7 2 1
```

The 7 would also then be swapped with the 2:

```
6 4 2 7 1
```

and then again with the 1 to give the state of the collection after the first whole pass along it:

```
6 4 2 1 7
```

The highest value will always end up at the end position after the first pass. This is where the algorithm gets its name – the highest value has "bubbled" to

the end. This also explains why the number of passes that will be needed can be forecast in advance: the number of values to be sorted[7].

In this example the second pass would see the 6 move to the position before the 7:

```
6 4 2 1 7
4 6 2 1 7   // 6 and 4 compared and swapped
4 2 6 1 7   // 6 and 2 compared and swapped
4 2 1 6 7   // 6 and 1 compared and swapped
4 2 1 6 7   // 6 and 7 compared and no swap needed
```

The final three passes would complete the sorting. Five passes are needed to sort this collection, which is as expected since there are five values to sort.

Bubblesort is not a very efficient sorting algorithm and this is not a particularly efficient implementation of it but it will serve to sort small numbers of values.

The implementation of this algorithm in C++ involves two `for` loops, one to count the number of passes required and one to count through the elements. Assuming that a vector of integers called *`numbers`* is being sorted into ascending order the loops are:

```
for (int pass = 0; pass < numbers.size (); pass ++) {
  for (int element = 0; element < numbers.size () - 1; element ++) {
```

Inside the loops the pairs of elements must be compared and swapped if needed. The swapping involves the use of a temporary variable to store one of the values:

```
int temp;

// Compare an element with the one to its right

if (numbers[elements] > numbers[elements + 1]) {

  // Swap them over if needed

  temp = numbers[elements];
  numbers[elements] = numbers[elements + 1];
  numbers[elements + 1] = temp;
}
```

If a vector is being sorted, it is the only parameter needed for a sorting function; an array of course requires that the number of elements is also passed. Since the sorting will very probably change both the array and vector, they must obviously be reference parameters (remember that arrays are always reference parameters and can only be passed as value parameters if declared as `const`). The prototypes for functions that would sort vectors or arrays of integers would be:

```
void sortVector (vector <int>&);
void sortArray (int[], int);
```

The complete function for sorting a vector of integers is below. This can be easily adapted to sort vectors of any other type. Apart from the obvious need to change the type of the parameter, the two things that need to be changed

7 Or is it actually *one less* than this number? Try it and see.

are the comparison and the type of the temporary variable used for swapping two values. Don't be afraid to adapt and use this code in your own programs[8]!

```
void sortVector (vector <int>& n)
{

  int temp;

  for (int p = 0; p < n.size (); p ++) {
    for (int e = 0; e < n.size () - 1; e ++) {
      if (n[e] > n[e + 1]) {
        temp = n[e];
        n[e] = n[e + 1];
        n[e + 1] = temp;
      } // if
    } // each pass
  } // total passes

  return;
}
```

One final point. The swapping of values in this function is a common operation, and we saw a function to do just that in the previous chapter:

```
// swap two integer values

void swap (int& a, int& b)
{
  int temp = a;

  a = b;
  b = temp;

  return;
}
```

The sorting function can make use of this function to produce a slightly neater and shorter version:

```
void sortVector (vector <int>& n)
{

  int temp;

  for (int p = 0; p < n.size (); p ++) {
    for (int e = 0; e < n.size () - 1; e ++) {
      if (n[e] > n[e + 1]) {
        swap (n[e], n[e + 1]);
      } // if
    } // each pass
  } // total passes

  return;

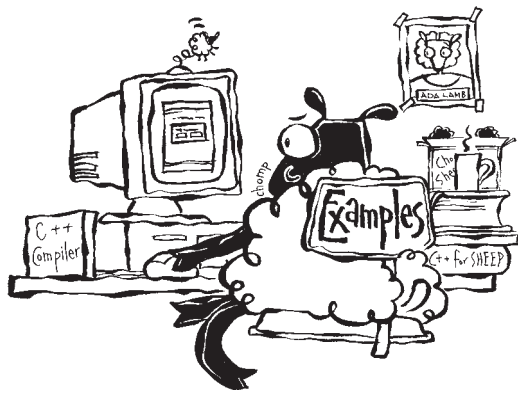
}
```

8 I confess that whenever I need to sort a collection of values I take down the nearest C++ book and seek out its definition of a Bubblesort. Feel free to use this one when you need one!

The *swap* function can easily be rewritten to handle values of another type.

Vectors and arrays

Vectors and arrays are notoriously difficult new concepts to grasp. It's important that you make sure that you understand them properly and that you are confident that you can use them. There are very few serious programs indeed that would not make use of one of these techniques. So it's time to get down to some practice with the examples and exercises!



Example 1 – Word fun

Flossy wants to practise using vectors. Mollie asks her to write a program that will process a list of words. It should find the total number of letters in all the words and the average length of the words, and should identify the longest and shortest words.

It should work as follows. The user should be prompted to enter a list of words, terminated with a single "X". There can be any number of words. If there is more than one word of the shortest or longest length any of them can be reported.

In fact, this program could be written without using vectors at all! It would be possible to write the whole program as a single great loop, making a set of comparisons and keeping totals every time a word is entered. This would, however, be extremely error-prone and would be very difficult to maintain. The program must be split up.

An obvious way of splitting the program into smaller units is to use one function to find each of the average length, the total number of letters, the longest word, and the shortest word. Each of these functions would take the collection of words as a parameter and would return the required value.

Since the number of words cannot be known in advance, a vector is the obvious choice to store the collection. The prototypes for the functions are then:

```
float averageLength (vector <string>);
int totalLetters (vector <string>);
string longestWord (vector <string>);
string shortestWord (vector <string>);
```

None of these functions will change the vector, so the parameters are all value parameters.

The program itself must first read the words into a vector:

```
vector <string> theWords;
```

using a loop that terminates when the user enters *X*. Each time a word is read, `push_back` is used to add it to the vector.

```
cout << "Enter Word (X to end): ";
cin >> tempWord;

while (tempWord != "X") {
  theWords.push_back (tempWord);
  cout << "Enter Word (X to end): ";
  cin >> tempWord;
}
```

The output is simply a case of calling each of the functions and displaying the results with some suitable labels.

The functions themselves are straightforward. Each is a case of examining each element of the vector in turn. For example, here is the function to find the total number of letters in the words:

```
int totalLetters (vector <string> w)
{

  int total = 0;

  for (int i = 0; i < w.size (); i ++) {
    total += w[i].length ();
  }

  return total;

}
```

The others are similar and are included in the complete program:

```
// words.cc
//
// Analyses a collection of words.
//
// Chapter 17, Examples 1.
//
// Author:    AMJ
// Date:      3rd January 2002
// Platform:  g++, Linux

#include <iostream>
#include <string>
#include <vector>

using namespace std;

float averageLength (vector <string>);
string longestWord (vector <string>);
string shortestWord (vector <string>);
int totalLetters (vector <string>);

int main ()
{

  vector <string> theWords;
```

```
    string tempWord;

    // get first word

    cout << "Enter Word (X to end): ";
    cin >> tempWord;

    while (tempWord != "X") {
      theWords.push_back (tempWord);
      cout << "Enter Word (X to end): ";
      cin >> tempWord;
    }

    // output

    if (theWords.size () > 0) {
      cout << "Longest Word:    " << longestWord (theWords)
           << endl
           << "Shortest Word:   " << shortestWord (theWords)
           << endl
           << "Words:           " << theWords.size ()
           << endl
           << "Total Letters:   " << totalLetters (theWords)
           << endl
           << "Average Length:  "
           << averageLength (theWords)
           << endl;
    }
    else {
      cout << "No words entered" << endl;
    }

    return 0;
  }

  float averageLength (vector <string> w)
  {

    int totalChars = 0;

    for (int i = 0; i < w.size (); i ++) {
      totalChars += w[i].length ();
    }

    return float (totalChars) / w.size ();

  }

  string longestWord (vector <string> w)
  {

    string longest = w[0];

    for (int i = 1; i < w.size (); i ++) {
      if (w[i].length () > longest.length ()) {
        longest = w[i];
      }
    }
```

```
    return longest;

  }

  string shortestWord (vector <string> w)
  {

    string shortest = w[0];

    for (int i = 1; i < w.size (); i ++) {
      if (w[i].length () < shortest.length ()) {
        shortest = w[i];
      }
    }

    return shortest;

  }

  int totalLetters (vector <string> w)
  {

    int total = 0;

    for (int i = 0; i < w.size (); i ++) {
      total += w[i].length ();
    }

    return total;

  }
```

Example 2 – Flossy's averages

Flossy's enthusiasm for playing cricket with her friends remains high. There have now been several matches and the sheep are taking a great deal of interest in their batting averages[9]*. There is much discussion about which sheep has the best average and Flossy has decided to write a program that will sort the averages into order.*

There are 11 sheep in the team. Flossy wants a program that will prompt her to enter the name and average of each sheep and will then display the team sorted into order of average with the highest average first.

Flossy's program clearly requires an array or vector to store the averages. In fact, two will be needed since vectors and arrays can store values of only one type. One will be needed for the names and one for the averages. Flossy will have to make sure that the elements in each correspond, so that, for example, the sheep whose name is at index 5 in the collection of names has its average at index 5 in the corresponding average collection[10]. This is quite a common situation.

---

9 It's the average number of runs scored in each innings.

10 A neater solution would be to implement a class that stored the sheep's name and the average (or even to write a program that used such a class if one already existed). The process used in this example is a common one where there are no classes or where there is no time or need to develop a class.

The choice of whether to use vectors and arrays is not straightforward. In this case the size of the collections can be predicted and is known in advance. It is also presumably unlikely to change. This tends to outweigh the usual advantages of using vectors, so arrays will be a good choice.

A constant integer will be declared to store the number of sheep in the team and will be used to declare the two arrays:

```
const int SheepInTeam = 11;

string names[SheepInTeam];
float averages[SheepInTeam];
```

A simple `for` loop can then be used to read the names and averages in, taking care that the elements of the two arrays correspond as required:

```
for (int i = 0; i < SheepInTeam; i ++) {
  cout << "Enter Name of Sheep #" << i + 1 << ": ";
  cin >> names[i];

  cout << "Enter " << names[i] << "\'s average: ";
  cin >> averages[i];
}
```

After the sorting has taken place, the results can be displayed using a similar loop. When writing programs like this it is always a good idea to make sure that the values are being read in and output correctly before starting to work on the sorting. There is, after all, very little point in trying to sort incorrect data!

For a small collection such as this a Bubblesort is a reasonable choice. Few changes are required to the usual form of the function. This time there are two arrays to be passed as parameters and the number of elements must also be included:

```
void sortTeam (string[], float[], int);
```

The sorting is based solely on the average, but, if a swap is required, the corresponding entries in both arrays must be swapped. The full function is included in the complete program. A final addition is a section of program to check that the average entered is not negative.

```
// sortSheep.cc
//
// Sorts a team of sheep into batting average order.
//
// Chapter 17, Examples 2.
//
// Author:    AMJ
// Date:      2nd January 2002
// Platform:  g++, Linux

#include <iostream>
#include <string>

using namespace std;

void sortTeam (string[], float[], int);

int main ()
```

```
{

  const int SheepInTeam = 11;

  string names[SheepInTeam];
  float averages[SheepInTeam];

  // get names and averages

  for (int i = 0; i < SheepInTeam; i ++) {
    cout << "Enter Name of Sheep #" << i + 1 << ": ";
    cin >> names[i];

    cout << "Enter " << names[i] << "\'s average: ";
    cin >> averages[i];

    while (averages[i] < 0.0) {
      cout << "Error! Average cannot be negative."
           << endl;
      cout << "Enter " << names[i] << "\'s average: ";
      cin >> averages[i];
    }
  }

  // sort

  sortTeam (names, averages, SheepInTeam);

  // output sheep

  for (int i = 0; i < SheepInTeam; i ++) {
    cout << names[i] << ": " << averages[i] << endl;
  }

  return 0;
}

void sortTeam (string n[], float a[], int num)
{

  string tempName;
  float tempAverage;

  for (int p = 0; p < num; p ++) {
    for (int e = 0; e < num - 1; e ++) {
      if (a[e] < a[e + 1]) {
        tempName = n[e];
        tempAverage = a[e];
        n[e] = n[e + 1];
        a[e] = a[e + 1];
        n[e + 1] = tempName;
        a[e + 1] = tempAverage;
      } // if
    } // each pass
  } // total passes

  return;
}
```

Of course, this program could also have been written using vectors. In this case the version using vectors is slightly more cumbersome, especially the sections that read the names and averages since extra temporary variables must be used. In this version there is no need to pass the number of sheep in the team into the sorting function.

For the sake of comparison the full program using vectors in this way is:

```
// sortSheep.cc
//
// Sorts a team of sheep into batting average order.
//
// Chapter 17, Examples 2.
//
// Author:    AMJ
// Date:      2nd January 2002
// Platform:  g++, Linux

#include <iostream>
#include <string>
#include <vector>

using namespace std;

void sortTeam (vector <string>&, vector <float>&);

int main ()
{

  const int SheepInTeam = 11;

  vector <string> names;
  vector <float> averages;

  string tempName;
  float tempAverage;

  // get names and averages

  for (int i = 0; i < SheepInTeam; i ++) {
    cout << "Enter Name of Sheep #" << i + 1 << ": ";
    cin >> tempName;

    cout << "Enter " << tempName << "\'s average: ";
    cin >> tempAverage;

    while (tempAverage < 0.0) {
      cout << "Error! Average cannot be negative."
           << endl;
      cout << "Enter " << tempName << "\'s average: ";
      cin >> tempAverage;
    }

    names.push_back (tempName);
    averages.push_back (tempAverage);
  }

  // sort
```

```
  sortTeam (names, averages);

  // output sheep

  for (int i = 0; i < names.size (); i ++) {
    cout << names[i] << ": " << averages[i] << endl;
  }

  return 0;
}

void sortTeam (vector <string>& n, vector <float>& a)
{

  string tempName;
  float tempAverage;

  for (int p = 0; p < n.size (); p ++) {
    for (int e = 0; e < n.size () - 1; e ++) {
      if (a[e] < a[e + 1]) {
        tempName = n[e];
        tempAverage = a[e];
        n[e] = n[e + 1];
        a[e] = a[e + 1];
        n[e + 1] = tempName;
        a[e + 1] = tempAverage;
      } // if
    } // each pass
  } // total passes

  return;

}
```

The need for temporary variables is rather messy in this example. It can be avoided by using vectors in a more "array-like" way by making use of the fact that the number of elements in the vector is known in advance (it is stored in the constant *SheepInTeam*). One final version of this program uses vectors in this way; it is probably the neatest of the three programs.

```
// sortSheep.cc
//
// Sorts a team of sheep into batting average order.
//
// Chapter 17, Examples 2.
//
// Author:    AMJ
// Date:      2nd January 2002
// Platform:  g++, Linux

#include <iostream>
#include <string>
#include <vector>

using namespace std;

void sortTeam (vector <string>&, vector <float>&);

int main ()
```

```
{

  const int SheepInTeam = 11;

  vector <string> names (SheepInTeam);
  vector <float> averages (SheepInTeam);

  // get names and averages

  for (int i = 0; i < SheepInTeam; i ++) {
    cout << "Enter Name of Sheep #" << i + 1 << ": ";
    cin >> names[i];

    cout << "Enter " << names[i] << "\'s average: ";
    cin >> averages[i];

    while (averages[i] < 0.0) {
      cout << "Error! Average cannot be negative."
           << endl;
      cout << "Enter " << names[i] << "\'s average: ";
      cin >> averages[i];
    }

  }

  // sort

  sortTeam (names, averages);

  // output sheep

  for (int i = 0; i < names.size (); i ++) {
    cout << names[i] << ": " << averages[i] << endl;
  }

  return 0;
}

void sortTeam (vector <string>& n, vector <float>& a)
{

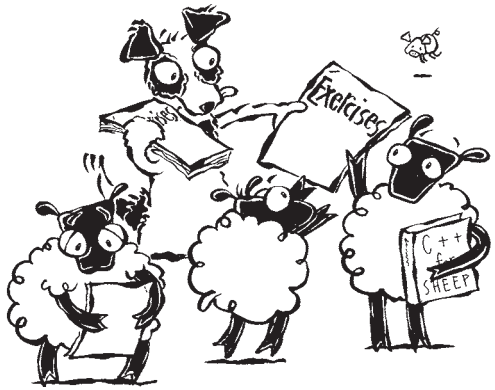
  string tempName;
  float tempAverage;

  for (int p = 0; p < n.size () - 1; p ++) {
    for (int e = 0; e < n.size () - 1; e ++) {
      if (a[e] < a[e + 1]) {
        tempName = n[e];
        tempAverage = a[e];
        n[e] = n[e + 1];
        a[e] = a[e + 1];
        n[e + 1] = tempName;
        a[e + 1] = tempAverage;
      } // if
    } // each pass
  } // total passes

  return;

}
```

These programs illustrate most of the common operations that make use of arrays and vectors. It would be good idea to go through them and to make sure that you understand how they work.



17.1 Examine the following code:

```
int numbers[6] = {1, 2, 3, 4, 5, 6};

cout << numbers[2];
cout << numbers[3] + numbers[0];
cout << numbers[6];
```

What is output by each of the three `cout` statements?

17.2 Examine the following code:

```
vector <int> numbers;

numbers.push_back (10);
numbers.push_back (20);
```

How many elements are there in the vector when it is declared? How many are there after this code has been executed. What value is stored in the element *`numbers[1]`*?

17.3 Write a program that prompts the user to enter the ages of 10 sheep and then displays the average age of the sheep. Your program should make use of this function to calculate the average:

```
float mean (int numbers[], int num)
{
  int total = 0;

  for (int i = 0; i < num; i ++) {
    total += numbers[i];
  }

  return float (total) / num;
}
```

17.4 Alter your program so that it can handle any number of sheep. The program should start by asking the user to enter the number of sheep. You will need to change the function.

17.5 A program is required to sort the following set of values into ascending order:

```
12 6 23 1 17 90
```

Using nothing more than pen and paper work carefully through the Bubblesort algorithm on this collection of values. You should know in advance how many times you will need to go through the values.

What does this example hint at about the efficiency of Bubblesort?

17.6 Using the algorithm given in this chapter and making some slight alterations to the Bubblesort code, write a program that will prompt the user to enter a collection of words and will display them in alphabetical order.

17.7 A simple *Sheep* class has the definition:

```
#include <string>

using namespace std;

class Sheep {
  private:
    string name;
    int age;

  public:
    Sheep ();

    // selectors

    string getName () const;
    int getAge () const;

    // mutators

    void setName (string);
    void setAge (int);

    // print name and age

    void print () const;

};
```

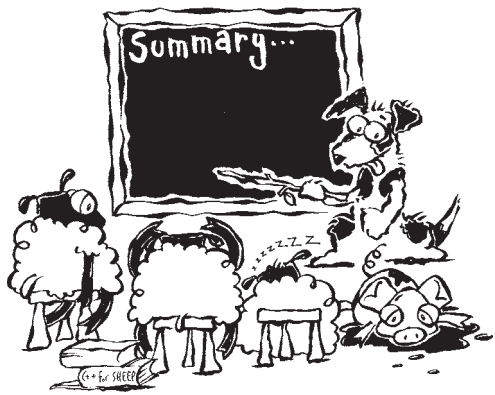
Implement this class and then use it in a program that prompts the user to enter the names and ages of a flock of sheep and then displays the details of the sheep (using the *print* method) in ascending order of age.

17.8 Extend your program so that it also displays the names of the oldest and youngest sheep. Assume that all the sheep have different ages.

17.9 Extend your program again so that each sheep can be entered only once. The program should check the existing names of the sheep each time a new one is entered and should display an error message if the sheep's details have been entered before. Assume that all the sheep have different names.

17.10 The implementation of Bubblesort in this chapter is not especially efficient. Work through the example again. Can you spot anything that is

particularly inefficient? Can you think of a way to make the existing implementation more efficient?



Vectors and arrays both allow programs to handle collections of data. All values in a collection must be of the same type, but can be of any available type, including classes.

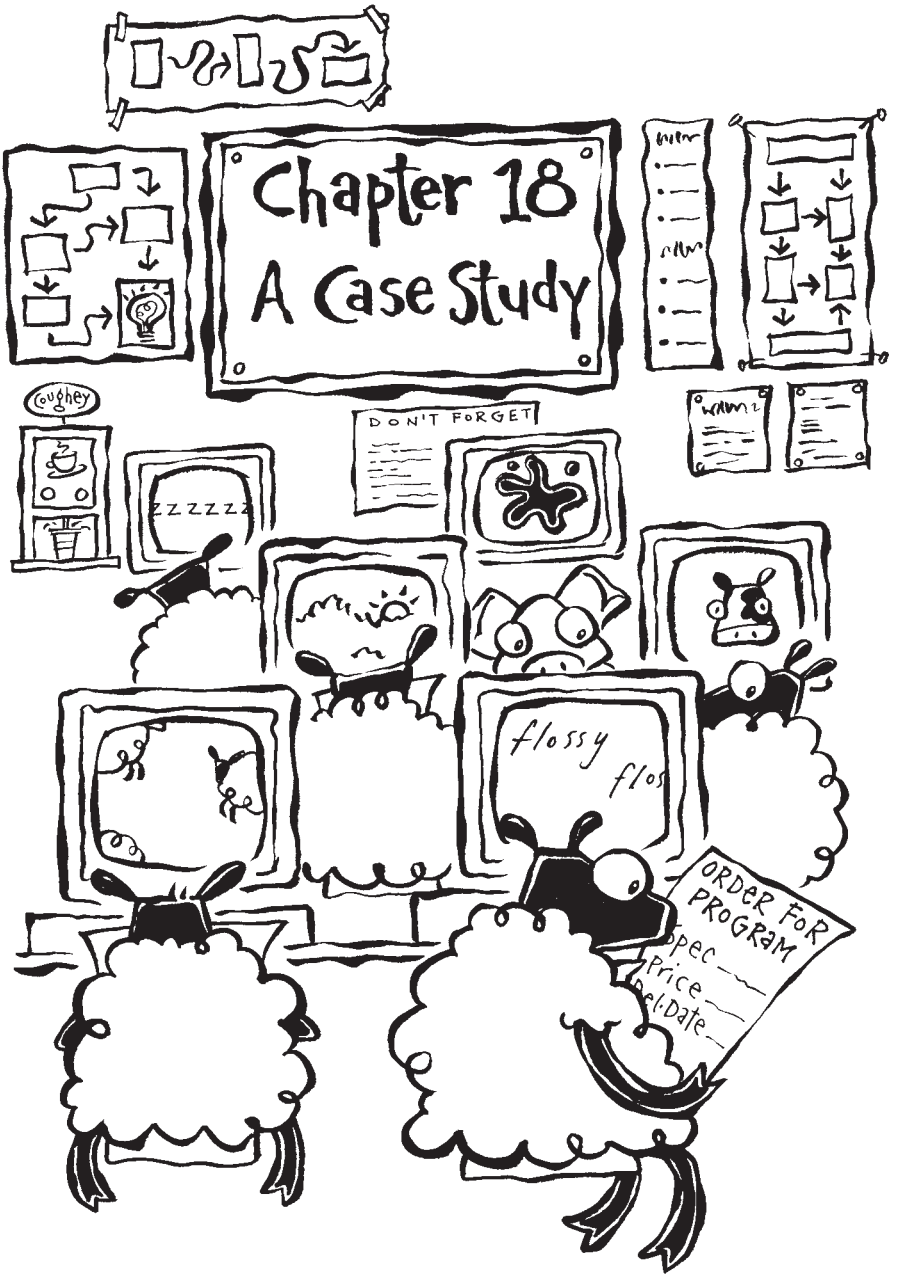
Arrays are inherited from C. Their size is fixed and cannot be changed after the array has been declared.

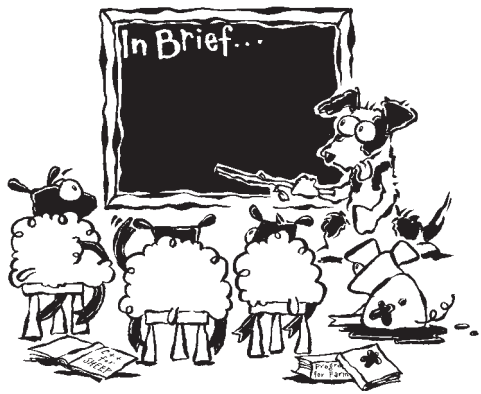
Vectors are specific to C++. Their size is dynamic and they can grow as more values are added. There is effectively no upper limit on the number of elements that can be stored in a vector.

For most applications vectors are rather neater to use than arrays. It is important to remember, though, that vectors are only available in C++, while arrays are more generally available in most programming languages. It should be obvious that vectors are essentially just a slightly more sophisticated version of arrays.

You should now be able to write programs that can handle collections of values. Specifically, you should be able to write programs that can sort these collections into some useful order. You should understand how to implement and use the Bubblesort sorting algorithm, and you should have some idea of why Bubblesort is a useful if not especially efficient approach to use for sorting.

That is the end of the C++ in this book! There is much more in C++ that you can hopefully now go on to learn about. Before finishing, there's a Case Study to tie everything together, and then there are some ideas about the things you might need or want to go on and learn now. Also, by now your programs are getting quite complicated so there's another very brief word about testing them.


Chapter 18
A Case Study
coughey
DON'T FORGET
ZZZZZZ
flossy
floss
ORDER FOR PROGRAM
Spec
Price
Del-Date

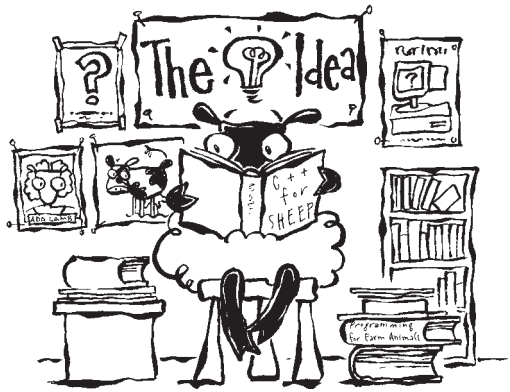


This is nearly the end. You have now seen, and had the chance to practice, all the C++ that there is in this book. This chapter presents some reasonably long programs that make use of all of this C++. It goes through the whole process of developing a complete object-oriented program; it starts with the description of a problem and then goes through the process of designing and developing suitable classes and then the programs that use them.

There have been many example programs in this book so far. All of them have been intended to illustrate the particular new part of C++ that was being described in the current chapter. The programs in this chapter draw from all of the C++ in the book. You should certainly be able to understand all these programs, and you should be able to write similar programs yourself.

After reading this chapter you should understand how a C++ program is written. You should know why the classes are designed and developed first and how the programs that make use of the classes are then designed and developed. You will also have seen once again some of the advantages of object-oriented programming and code reuse.

This chapter will also in passing highlight some of the areas that have not been covered. These are for you to find out about later!



Farmer Strawstrap has been becoming increasingly interested in the programming antics of his animals. As they have mastered more and more C++

the animals have decided to set up their own software development company, and Farmer Strawstrap has agreed to be their first customer.

The farmer's interests are already well known. He is interested in keeping details of all his animals and in particular their values. He is also keen to introduce a tracking system so that he knows where each animal is at any moment; there is a nasty big dark hole in one of his fields, and he is keen not to lose any animals down it, least of all the valuable ones.

The approach

The farmer has a lot of requirements and will eventually want many programs to solve many problems. The object-oriented approach to a set of problems such as this is obviously to develop the classes that his programs will need first and then to develop the programs themselves. The classes will be used in many programs, a fine example of code reuse.

Some further analysis of the farmer's requirements is needed. The first step is to identify the object types that are needed and the attributes and methods of each. It should be reasonably easy to design and implement some suitable classes from the results of this analysis.

The requirements

There are sheep and pigs on the farm and Farmer Stroustrup is keen to store details of both. There is actually also one cow on the farm, but this creature is big enough for the farmer to see all the time and she rarely strays very far from the farmhouse[1]. The first programs will require only information about sheep and pigs; the cow can always be added later.

For each sheep and pig the farmer needs to store the name, age, weight, value and current position on the farm. His tracking system can provide a simple pair of integers as coordinates for each animal's position.

The value of the animals is of particular interest and the farmer has devoted no little time to devising incredibly complex formulas that tell him the maximum and minimum value for each animal. This is most complex for sheep, whose maximum value is given by these steps:

```
THE MAXIMUM VALUE IS 1.8 TIMES THE ACTUAL VALUE.
IF THE SHEEP IS AGED MORE THAN 5 EACH YEAR OVER 5 REMOVES 0.80 FROM
  THE VALUE.
BLACK SHEEP ARE WORTH 2.0 MORE THAN OTHER SHEEP.
```

Happily the other values are simpler. The maximum value of a pig is just 1.5 times the actual value and the minimum values for both types of animal are simply half the maximum value.

Initially the farmer would like three programs. The first will provide him with a list of the sheep sorted into order by maximum value with the most valuable listed first. The second will deal with the problem of the hole in the field; it will check whether any animals seem to have fallen down the hole and will report which sheep and pig is nearest to it. Finally a program is required that will determine which is the most valuable animal on the farm.

---

1 In any case, she is significantly larger than the big hole. If you've been reading carefully you might have realised that the big hole and the milking shed are at the same coordinates. In case you were worrying, they're in different fields.

Armed with this information the first step is to identify and design the classes that will be needed.

The classes

Identifying the classes appears to be easy. At first sight it appears that there is obviously a need for an *Animal* class that will store the information required. However, looking a little more closely at the details of the requirements reveals that this might not be quite the case.

The formula for working out the maximum value for a sheep reveals that sheep of a certain colour have different values. If this calculation is to be implemented then the colour of the sheep will also have to be stored as an attribute of the class. This does not seem to be the case with pigs, whose colour[2] appears to be quite irrelevant to their value. There will have to be two classes, then – one for sheep (*Sheep*) and one for pigs (*Pig*).

The attributes of the *Sheep* class are easy to identify; all that remains is to assign data types to them. Name and colour are strings (so the definition will have to include the `string` library), age and weight are integers, and the value is a floating-point number. The sheep's coordinates are integers. These assumptions should really be checked with the farmer, who might want to store weights more accurately as floating-point numbers, for example, but they will do.

The methods are more complex. Obviously each attribute will need a selector and a mutator[3], but appropriate methods will need to be added so that the required programs can be implemented. Also, it is probably a good idea to add methods that will become useful in future programs, even if they are not needed in the three that are required first.

When identifying the methods it is important to separate them from the functions that will be needed in the programs. A method is attached to a single instance of a class, while non-member functions in programs may well process collections of instances. An example of that here is in the first program, to sort the sheep into order of value. A sorting function will obviously be needed but this is part of the program and not part of the class; it will process a collection of sheep. The only method needed for this sorting process will be one to return the maximum value of an individual sheep, since this is used as the basis for the sort.

The problem of the hole relates to single sheep, so methods in the class are appropriate. It means that two methods will be needed: one to determine whether the sheep is in the hole and one to calculate how far away from the hole the sheep is. These methods can be made more generally useful by making them deal with an arbitrary point rather than the specific coordinates of the hole[4]; the coordinates of the point on the farm to be considered can be provided by parameters.

The problem of tracking the sheep is familiar by now. Methods will be needed to move a sheep to a particular coordinate and to move the sheep in any of the four compass directions.

---

2 Muddy, presumably.

3 It is possible that the programs won't need all of these, but there are sufficiently few attributes that they might as well all have a selector and a mutator.

4 All right, so the hole won't move, but there could be *other holes* out there.

All the programs will involve much input and output of the attributes. It makes sense to implement a method to provide all the prompts to input these values and then read them and another to output them all in some neat format.

The attributes and methods for the *Pig* class are the same as for the *Sheep* class except that there is no attribute for the pig's colour. This will inevitably lead to some methods being identical in the two classes, but it will at least mean that programs that deal with just one sort of animal can be written more neatly. In any case a single class to store details of both types of animal would also need some sort of "type of animal" attribute.

Defining the Sheep class

With the attributes and methods identified, writing the definition file for the *Sheep* class is straightforward. The private section is simply a list of the attributes:

```
private:
  string name;
  string colour;
  int age;
  int weight;
  float value;

  int x;
  int y;
```

The public interface includes a constructor and starts with the selectors and mutators:

```
public:
  Sheep ();

  // selectors

  string getName () const;
  string getColour () const;
  int getAge () const;
  int getWeight () const;
  float getValue () const;

  int getX () const;
  int getY () const;

  // mutators

  void setName (string);
  void setColour (string);
  void setAge (int);
  void setWeight (int);
  void setValue (float);

  void setX (int);
  void setY (int);
```

The methods to determine the maximum and minimum value look rather like selectors, and indeed a programmer using them without any knowledge

of the private attributes could think that they were. They come next together with a third method to return the range:

```
// derived values

float getMaxValue () const;
float getMinValue () const;
float getValueRange () const;
```

This last method is not needed for any of the first three programs, but it is simple to write and may well come in useful in the future.

The problem of recording the sheep's position and any changes in it brings in the methods to move the sheep around the farm. There are also the methods to determine whether the sheep is at a particular point and the distance from a particular point. These will be used in the program that will handle the problem of the hole in the field but they may well also be more generally useful elsewhere later.

```
// Move in compass directions

void moveNorth ();
void moveSouth ();
void moveEast ();
void moveWest ();

// Move to specific point

void moveTo (int, int);

// True if at postition (x, y) in parameters

bool atPosition (int, int) const;

// Distance from point (x, y) in parameters

float distanceFrom (int, int) const;
```

Finally there are methods for input and output. The first is a method to prompt the user to input the values for an instance of the class. This will be used in many programs and implementing it in the class will save a lot of potentially duplicated work. A method to display the sheep's location will be useful, as will methods to display a summary and full details of the sheep's information.

```
// input from keyboard

void input ();

// Neatly formatted output in form (x, y)

void printLocation () const;

// output to screen

void outputSummary () const;
void outputComplete () const;
```

A full listing of the header file can be found with the rest of the files from this case study on the web site. The methods that do not change any of the values of any of the attributes have all had a `const` added to the prototype.

The header file for the *Pig* class is much the same with the omission of the selector and mutator for colour. The two classes will have different implementations for some of the methods.

Implementing the Sheep class

Most of the methods are only a few lines long and are trivial to implement; this is an advantage of splitting the program into very small parts. The form of the selectors and mutators should be familiar, for example:

```
// selectors

string Sheep::getName () const
{
  return name;
}

// mutators

void Sheep::setName (string n)
{
  name = n;
  return;
}
```

The methods to record changes in the sheep's position are similarly short:

```
// Move in compass directions

void Sheep::moveNorth ()
{
  y ++;
  return;
}

// Move to specific point

void Sheep::moveTo (int newX, int newY)
{
  x = newX;
  y = newY;

  return;
}
```

The two methods to determine whether a sheep is at a certain point or how far away the sheep is are both only one line, even if the line is quite complex. The first is simply a case of returning a condition:

```
// True if at postition (x, y) in parameters

bool Sheep::atPosition (int atX, int atY) const
{
  return x == atX && y == atY;
}
```

The second involves a calculation using Pythagoras' theorem. This includes a use of the built-in square root function, so the implementation will need to include the `cmath` library. The expression is quite complex and requires a lot of brackets to make sure that the calculation happens in the correct order. It is also necessary to add a cast, since `sqrt` can process only floating-point values[5].

```
// Distance from point (x, y) in parameters

float Sheep::distanceFrom (int atX, int atY) const
{
  return sqrt (float (((atX - x) * (atX - x))
                    + ((atY - y) * (atY - y))));
}
```

It would perhaps be better to make the method slightly longer in order to make the calculation clearer:

```
float Sheep::distanceFrom (int atX, int atY) const
{
  int xDiff = atX - x;
  int yDiff = atY - y;

  return sqrt (float ((xDiff * xDiff) + (yDiff * yDiff)));
}
```

This version includes brackets round the two multiplications that are not strictly necessary. They do serve to make the calculation as clear as possible to anyone reading the program, though. Sometimes in a complex expression it is a good idea to add unnecessary brackets in the interests of clarity.

The calculation of the maximum possible value of a sheep is more complex, but is quite simple when broken down. A local variable is used to store the value; it is initially assumed to be 1.8 times the actual value, and then changes are made if the sheep is particularly old or an expensive colour.

```
float Sheep::getMaxValue () const
{
  float v = value;

  v *= 1.8;

  if (age > 5) {
    v -= (age - 5) * 0.8;
  }

  if (colour == "black") {
    v += 2.0;
  }

  return v;
}
```

The minimum value relies on this maximum value so that method must first call the method to find the maximum value. That aside, the calculation is

5 You might well find that the version of `sqrt` on your system can handle integer values. Some versions can't.

trivial. Similarly, the method to find the range of possible values calls both these methods.

```
float Sheep::getMinValue () const
{
  return getMaxValue () / 2.0;
}

float Sheep::getValueRange () const
{
  return getMaxValue () - getMinValue ();
}
```

The method to input values for the private attributes is quite long, but at least this is the only place it will have to be written. Some loops are used to make sure that the age, weight and value are not negative.

```
void Sheep::input ()
{
  cout << "Enter name: ";
  cin >> name;

  cout << "Enter colour: ";
  cin >> colour;

  cout << "Enter age: ";
  cin >> age;

  while (age < 0) {
    cout << "Error - age cannot be negative" << endl;
    cout << "Enter age: ";
    cin >> age;
  }

  cout << "Enter weight: ";
  cin >> weight;

  while (weight < 0) {
    cout << "Error - weight cannot be negative" << endl;
    cout << "Enter weight: ";
    cin >> weight;
  }

  cout << "Enter value: ";
  cin >> value;

  while (value < 0) {
    cout << "Error - value cannot be negative" << endl;
    cout << "Enter value: ";
    cin >> value;
  }

  cout << "Enter X position: ";
  cin >> x;
  cout << "Enter Y position: ";
  cin >> y;

  return;
}
```

This method has assumed that the name and colour need no validation, and similarly that any coordinates are valid.

Finally, there are three methods to output the details of the sheep in various formats. This does not stop a programmer implementing another in a program, of course; the values can still be accessed directly via the selectors. The point of these methods is to provide generic output that can be used in many situations.

The first takes care of printing out the coordinates of the sheep with suitable brackets and a comma:

```
// Neatly formatted output in form (x, y)

void Sheep::printLocation () const
{
  cout << '(' << x << ", " << y << ')';
  return;
}
```

and the other two provide a summary or full display of the values:

```
// output to screen
// Format is: Flossy, value 100 is at (5, 3)

void Sheep::outputSummary () const
{
  cout << name << ", value " << value << " is at ";
  printLocation ();
  cout << endl;

  return;
}

void Sheep::outputComplete () const
{
  cout << "Name:      " << name << endl;
  cout << "Colour:    " << colour << endl;
  cout << "Age:       " << age << endl;
  cout << "Weight:    " << weight << endl;
  cout << "Value:     " << value << endl;
  cout << "Position:  ";
  printLocation ();
  cout << endl ;

  return;
}
```

The full implementation of the class is at the end of the chapter.

Implementing the `Pig` class

Almost all of the implementation of the *Pig* class is identical to the *Sheep* class. The only difference comes in the implementation of the method for maximum value. This is much simpler:

```
// derived values

float Pig::getMaxValue () const
{
```

```
    return value * 1.5;
}
```

Two classes can have methods with the same names, and the implementations can be different. The correct implementation will be found depending of the type of object used with the method. The full definition and implementation of the *Pig* class are on the web site.

Testing the classes

The next stage in the development would be to write small driver programs to test both the classes. These programs should test all the methods (paying particular attention to the methods dealing with the complexities of the values) so that the programmers can be sure that they work correctly.

Driver programs may simply call the methods of a class one after another, reporting the results, or they may be more complex. A driver for a sophisticated class would probably provide some sort of menu that could be used. There is more about driver programs and testing in the next chapter. For the moment, you can find the menu-based program that was used to test the *Sheep* class in this example on the web site.

The programs

With the classes written and tested it is time to start work on the programs. Some of the programs may well be many lines long, so it will be necessary to split them into functions. It will also be a good idea to be on the lookout for non-member functions that might be useful in more than one of the programs.

Sorting valuable sheep

The algorithm for this program is quite simple:

```
READ THE LIST OF SHEEP
SORT THE SHEEP ON MAXIMUM VALUE
OUTPUT THE SORTED LIST
```

The program will make use of only one of the classes, but it will be worth bearing in mind that the two classes are so similar that a program written to use one will need only a few modifications to work with the other.

A look at the other programs to be written reveals that reading a collection of sheep from the user is going to be a common task since the same operation is required in the other two programs. It therefore makes good sense to implement this operation as a non-member function. The function can be used in the other programs, and this will shorten the time needed to develop those.

The program needs to store and process a collection of sheep. Any number of sheep can be entered, so a vector is clearly the best choice for storage. This can be made to grow to the appropriate size as the data is entered. This vector will be passed as a reference parameter to the function that will read the values. The prototype for this function is therefore:

```
void readSheep (vector <Sheep>&);
```

There will be a corresponding function to write out the sorted list of sheep. In this case the vector does not need to be a reference parameter and can instead be passed as a value parameter:

```
void writeSheep (vector <Sheep>);
```

or perhaps as a constant reference:

```
void writeSheep (const vector <Sheep>&);
```

This program is also going to involve some sorting, but before starting on this it is sensible to implement and test these two functions. There is no point at all in working on the sorting function if the function to read the values to be sorted does not work and if the results cannot be seen!

The code to read the details of a single sheep is very short, making use of the *input* method provided. A temporary class instance is needed so that it can be added to the vector:

```
vector <Sheep> theSheep;  // the list of sheep
Sheep tempSheep;          // a temporary sheep

cout << "Enter details of the first sheep: " << endl;
tempSheep.input ();
theSheep.push_back (temp);
```

The *readSheep* function then requires a simple dialogue to prompt the user to enter values until they have finished. In full this function reads:

```
void readSheep (vector <Sheep>& s)
{

  Sheep temp;

  char answer;

  cout << "Enter details of the first sheep: " << endl;
  temp.input ();
  s.push_back (temp);

  cout << "Enter another? (y/n): ";
  cin >> answer;

  answer = tolower (answer);

  while (answer != 'y' && answer != 'n') {
    cout << "Please enter \'y\' or \'n\' ..." << endl;
    cout << "Enter another? (y/n): ";
    cin >> answer;
  }

  while (answer == 'y') {
    cout << "Enter details of the next sheep: " << endl;
    temp.input ();
    s.push_back (temp);

    cout << "Enter another? (y/n): ";
    cin >> answer;

    answer = tolower (answer);

    while (answer != 'y' && answer != 'n') {
      cout << "Please enter \'y\' or \'n\' ..." << endl;
      cout << "Enter another? (y/n): ";
      cin >> answer;
```

```
    }
  }

  return;
}
```

The coding of the dialogue has been simplified in this function with some cunning use of the `tolower` function to convert the case of characters. The user's answer to the prompt is converted to lower case, so the comparison statements are rather shorter.

The function to write a sorted list is a simple `for` loop. For the moment this can just output the name and value; presumably this could be refined if required.

```
void writeSheep (vector <Sheep> s)
{
  for (int i = 0; i < s.size (); i ++) {
    cout << s[i].getName () << ", value "
         << s[i].getMaxValue () << endl;
  }

  return;
}
```

The first version of the main program simply calls these two functions. The program itself involves very few lines, which is quite common. The use of functions and classes often means that main programs are very short.

The program includes the required libraries and class definition, provides the prototypes of the two functions, and declares the vector. The vector is then used by the two functions.

```
#include <cctype>   // for tolower ()
#include <iostream>
#include <vector>

#include "sheep.h"

using namespace std;

void readSheep (vector <Sheep>&);
void writeSheep (vector <Sheep>);

int main ()
{

  vector <Sheep> theSheep;

  // get the sheep

  readSheep (theSheep);

  // write the sorted list

  writeSheep (theSheep);

  return 0;

}
```

This program should be properly tested before any work is done on the sorting function. If this program is working correctly it should do nothing more than display the same data as was entered in the same order.

When this first program is correct the sorting can be implemented. A Bubblesort is as good a choice as any. Many programmers like to implement the part of the sorting function that swaps two values as a separate function (we have seen a function to swap two values before). This means that the sorting function itself is a little shorter and makes it more likely that it will work unchanged with data of other types.

This program will therefore have two additional functions, one to do the sorting and one to swap two sheep over:

```
void sheepSort (vector <Sheep>&);
void swap (Sheep&, Sheep&);
```

The swapping function makes use of the usual temporary variable:

```
void swap (Sheep& one, Sheep& two)
{
  Sheep temp;

  temp = one;
  one = two;
  two = temp;

  return;
}
```

and the sorting function uses the now familiar loops:

```
void sheepSort (vector <Sheep>& s)
{
  for (int p = 0; p < s.size () - 1; p ++) {
    for (int e = 0; e < s.size () - 1; e ++) {
      if (s[e].getMaxValue () < s[e + 1].getMaxValue ()) {
        swap (s[e], s[e + 1]);
      }
    }
  }

  return;
}
```

The program is now complete. All that remains is to add the prototypes for the two sorting functions to the main program and to call the *sheepSort* function in the main program. The full program is:

```
// sheepValue.cc
//
// Sorts sheep based on their maximum value.
//
// Chapter 18.
//
// Author:    AMJ
// Date:      5th January 2002
// Platform:  g++, Linux

#include <cctype>
#include <iostream>
```

```
#include <vector>

#include "sheep.h"

using namespace std;

void readSheep (vector <Sheep>&);
void writeSheep (vector <Sheep>);

void sheepSort (vector <Sheep>&);
void swap (Sheep&, Sheep&);

int main ()
{

  vector <Sheep> theSheep;

  // get the sheep

  readSheep (theSheep);

  // sort

  sheepSort (theSheep);

  // write the sorted list

  writeSheep (theSheep);

  return 0;

}

void readSheep (vector <Sheep>& s)
{

  Sheep temp;

  char answer;

  cout << "Enter details of the first sheep: " << endl;
  temp.input ();
  s.push_back (temp);

  cout << "Enter another? (y/n): ";
  cin >> answer;

  answer = tolower (answer);

  while (answer != 'y' && answer != 'n') {
    cout << "Please enter \'y\' or \'n\' ..." << endl;
    cout << "Enter another? (y/n): ";
    cin >> answer;
  }

  while (answer == 'y') {
    cout << "Enter details of the next sheep: " << endl;
    temp.input ();
    s.push_back (temp);
```

```
    cout << "Enter another? (y/n): ";
    cin >> answer;

    answer = tolower (answer);

    while (answer != 'y' && answer != 'n') {
      cout << "Please enter \'y\' or \'n\' ..." << endl;
      cout << "Enter another? (y/n): ";
      cin >> answer;
    }
  }

  return;
}

void writeSheep (vector <Sheep> s)
{
  for (int i = 0; i < s.size (); i ++) {
    cout << s[i].getName () << ", value "
         << s[i].getMaxValue () << endl;
  }

  return;
}

void sheepSort (vector <Sheep>& s)
{
  for (int p = 0; p < s.size () - 1; p ++) {
    for (int e = 0; e < s.size () - 1; e ++) {
      if (s[e].getMaxValue () < s[e + 1].getMaxValue ()){
        swap (s[e], s[e + 1]);
      }
    }
  }

  return;
}

void swap (Sheep& one, Sheep& two)
{

  Sheep temp;

  temp = one;
  one = two;
  two = temp;

  return;
}
```

The main function of this program is very short indeed. All it does is declare the required vector and then call the functions. Writing programs in this way can produce a collection of functions that are useful in other programs that use the same classes.

The big dark hole

There are two methods available that will obviously be of use in this program. The *`atPosition`* method will allow the program to determine whether

an animal is at the same position as the hole and the *distanceFrom* method will be used to find the animal that is closest.

The algorithm for the program is:

```
READ THE PIGS
READ THE SHEEP
CHECK IF ANY PIGS ARE AT THE HOLE
CHECK IF ANY SHEEP ARE AT THE HOLE
FIND THE CLOSEST PIG TO THE HOLE
FIND THE CLOSEST SHEEP TO THE HOLE
```

As before, there is clearly going to be a great deal of very similar code to deal with the two types of animal. The two collections of animals will be held in two vectors and will be processed in the same way.

A function to read a vector of sheep was written for the first program and it can be reused unchanged in this. A similar function to read the vector of pigs can be written by making a few simple changes to the *readSheep* function. This gives two function prototypes:

```
void readPigs (vector <Pig>&);
void readSheep (vector <Sheep>&);
```

There is no need to write out the complete vectors this time, so no corresponding *write* functions will be needed.

The methods to check the animals' position require that the position of the hole be passed as parameters. To make the program more readable and to provide for the (remote) possibility that the hole will move the coordinates are defined in the main function as constants:

```
const int DarkHoleX = 5;
const int DarkHoleY = 3;
```

The check to see if any animals are in the hole is a `for` loop calling *atPosition* on each animal in turn. Two very similar loops are needed, one for pigs and one for sheep. Assuming that the vector has the identifier *thePigs*, the loop for the pigs is:

```
for (int i = 0; i < thePigs.size (); i ++) {
  if (thePigs[i].atPosition (DarkHoleX, DarkHoleY)) {
    cout << "Pig in the hole: ";
    thePigs[i].outputSummary ();
  }
}
```

This loop could either be included in the main function as it stands or it could be implemented as a function:

```
void printPigsInHole (vector <Pigs>, int, int);
```

and called from the main function. The choice would depend on whether there seemed to be any chance of needing to use the same loop again in another program.

The process of finding the closest animals to the hole also uses a similar loop to go through each class instance in turn. This can be implemented as two functions, one for each type of animal, that return the closest class instance:

```
Pig closestPig (vector <Pig>, int, int);
Sheep closestSheep (vector <Sheep>, int, int);
```

These can be called from the main function. Since they return an instance of the class, the method *outputSummary* can be used to print the details in one step:

```
cout << "Closest pig is: ";
closestPig (thePigs, DarkHoleX, DarkHoleY).outputSummary ();
cout << "Closest sheep is: ";
closestSheep (theSheep, DarkHoleX, DarkHoleY).outputSummary ();
```

The first two of these lines have exactly the same effect as:

```
Pig tempPig;
cout << "Closest pig is: ";
tempPig = closestPig (thePigs, DarkHoleX, DarkHoleY);
tempPig.outputSummary ();
```

Some programmers would prefer to use this longer form to make the program clearer and easier to follow.

All the components of the program are now complete. In full it is:

```
// darkHole.cc
//
// Checks if any animals are near the dark hole.
//
// Chapter 18.
//
// Author:    AMJ
// Date:      5th January 2002
// Platform:  g++, Linux

#include <cctype>
#include <iostream>
#include <vector>

#include "pig.h"
#include "sheep.h"

using namespace std;

void readPigs (vector <Pig>&);
void readSheep (vector <Sheep>&);

Pig closestPig (vector <Pig>, int, int);
Sheep closestSheep (vector <Sheep>, int, int);

int main ()
{

  const int DarkHoleX = 5;
  const int DarkHoleY = 3;

  vector <Pig> thePigs;
  vector <Sheep> theSheep;

  // get the animals

  readPigs (thePigs);
  readSheep (theSheep);

  // check if any are in the hole
```

```
    for (int i = 0; i < thePigs.size (); i ++) {
      if (thePigs[i].atPosition (DarkHoleX, DarkHoleY)) {
        cout << "Pig in the hole: ";
        thePigs[i].outputSummary ();
      }
    }

    for (int i = 0; i < theSheep.size (); i ++) {
      if (theSheep[i].atPosition (DarkHoleX, DarkHoleY)) {
        cout << "Sheep in the hole: ";
        theSheep[i].outputSummary ();
      }
    }

    // find the closest animals

    cout << "Closest pig is: ";
    closestPig (thePigs, DarkHoleX, DarkHoleY).outputSummary ();
    cout << "Closest sheep is: ";
    closestSheep (theSheep, DarkHoleX, DarkHoleY).outputSummary ();

    return 0;

  }

  void readSheep (vector <Sheep>& s)
  {

    Sheep temp;

    char answer;

    cout << "Enter details of the first sheep: " << endl;
    temp.input ();
    s.push_back (temp);

    cout << "Enter another? (y/n): ";
    cin >> answer;

    answer = tolower (answer);

    while (answer != 'y' && answer != 'n') {
      cout << "Please enter \'y\' or \'n\' ..." << endl;
      cout << "Enter another? (y/n): ";
      cin >> answer;
    }

    while (answer == 'y') {
      cout << "Enter details of the next sheep: " << endl;
      temp.input ();
      s.push_back (temp);

      cout << "Enter another? (y/n): ";
      cin >> answer;

      answer = tolower (answer);

      while (answer != 'y' && answer != 'n') {
        cout << "Please enter \'y\' or \'n\' ..." << endl;
```

```
        cout << "Enter another? (y/n): ";
        cin >> answer;
      }
    }

    return;
  }

  void readPigs (vector <Pig>& p)
  {

    Pig temp;

    char answer;

    cout << "Enter details of the first pig: " << endl;
    temp.input ();
    p.push_back (temp);

    cout << "Enter another? (y/n): ";
    cin >> answer;

    answer = tolower (answer);

    while (answer != 'y' && answer != 'n') {
      cout << "Please enter \'y\' or \'n\' ..." << endl;
      cout << "Enter another? (y/n): ";
      cin >> answer;
    }

    while (answer == 'y') {
      cout << "Enter details of the next pig: " << endl;
      temp.input ();
      p.push_back (temp);

      cout << "Enter another? (y/n): ";
      cin >> answer;

      answer = tolower (answer);
      while (answer != 'y' && answer != 'n') {
        cout << "Please enter \'y\' or \'n\' ..." << endl;
        cout << "Enter another? (y/n): ";
        cin >> answer;
      }
    }

    return;

  }

  Pig closestPig (vector <Pig> p, int holeX, int holeY)
  {
    Pig closest = p[0];
    float distance = p[0].distanceFrom (holeX, holeY);

    for (int i = 0; i < p.size (); i ++) {
      if (p[i].distanceFrom (holeX, holeY) < distance) {
        closest = p[i];
        distance = p[i].distanceFrom (holeX, holeY);
```

```
    }
  }

  return closest;
}

Sheep closestSheep (vector <Sheep> s, int holeX, int holeY)
{
  Sheep closest = s[0];
  float distance = s[0].distanceFrom (holeX, holeY);

  for (int i = 0; i < s.size (); i ++) {
    if (s[i].distanceFrom (holeX, holeY) < distance) {
      closest = s[i];
      distance = s[i].distanceFrom (holeX, holeY);
    }
  }

  return closest;
}
```

The most valuable animal

The final program involves comparing the values of sheep and pigs. There is no direct way of doing this, so a more general solution will be needed. The basic algorithm for this program is not dissimilar to the others:

```
READ THE PIGS
READ THE SHEEP
FIND THE MOST VALUABLE ANIMAL
```

Functions have already been written for the first two tasks and these can be used unchanged in this new program; yet another fine example of code reuse.

New code will be needed for the final step to find the most valuable animal. There is no obvious way of doing this at the moment. Looking again at the problem it seems that Farmer Strawstrap is more than a little obsessed with the values of the animals. It seems reasonable to suppose that at some point in the future he will want to know which is his most valuable sheep, so a function to determine this would eventually be of use. This can be used in an implementation of this program. The same probably applies to pigs. Two functions will be defined to find the most valuable sheep and pig:

```
Sheep mostValuableSheep (vector <Sheep>);
Pig mostValuablePig (vector <Pig>);
```

and a third function will use the results of these two to find the most valuable animal overall. Its prototype will be:

```
void mostValuable (vector <Pig>, vector <Sheep>);
```

Finding the most valuable sheep is the usual case of using a loop to go through the vector, comparing the values. The function works by assuming that the first sheep is the most valuable, and then comparing the values of all the others.

```
Sheep mostValuableSheep (vector <Sheep> s)
{
  Sheep bestSheep = s[0];
```

```
    float bestValue = s[0].getValue ();

    for (int i = 0; i < s.size (); i ++) {
      if (s[i].getValue () > bestValue) {
        bestSheep = s[i];
        bestValue = s[i].getValue ();
      }
    }

    return bestSheep;
  }
```

The function to find the most valuable animal overall calls this function and the similar one for pigs and compares the results. This function cannot return the result, as it's impossible to know the type in advance; it must display the results itself. The implementation is:

```
void mostValuable (vector <Pig> p, vector <Sheep> s)
{
  Pig topPig = mostValuablePig (p);
  Sheep topSheep = mostValuableSheep (s);

  cout << "The most valuable animal is: " << endl;
  if (topPig.getValue () > topSheep.getValue ()) {
    topPig.outputComplete ();
  }
  else {
    topSheep.outputComplete ();
  }

  return;
}
```

The other functions have already been written, so only the main function remains. It is very short as before. It simply declares the two vectors, uses the functions to read the values in and then calls the function to display the most valuable animal. The complete program is:

```
// valuableAnimal.cc
//
// Finds the most valuable animal - sheep or pig!
//
// Chapter 18.
//
// Author:    AMJ
// Date:      5th January 2002
// Platform:  g++, Linux

#include <cctype>
#include <iostream>
#include <vector>

#include "pig.h"
#include "sheep.h"

using namespace std;

void readPigs (vector <Pig>&);
void readSheep (vector <Sheep>&);
```

```
Sheep mostValuableSheep (vector <Sheep>);
Pig mostValuablePig (vector <Pig>);

void mostValuable (vector <Pig>, vector <Sheep>);

int main ()
{

  vector <Pig> thePigs;
  vector <Sheep> theSheep;

  // get the animals

  readPigs (thePigs);
  readSheep (theSheep);

  // display the most valuable

  mostValuable (thePigs, theSheep);

  return 0;
}

void readSheep (vector <Sheep>& s)
{

  Sheep temp;

  char answer;

  cout << "Enter details of the first sheep: " << endl;
  temp.input ();
  s.push_back (temp);

  cout << "Enter another? (y/n): ";
  cin >> answer;

  answer = tolower (answer);

  while (answer != 'y' && answer != 'n') {
    cout << "Please enter \'y\' or \'n\' ..." << endl;
    cout << "Enter another? (y/n): ";
    cin >> answer;
  }

  while (answer == 'y') {
    cout << "Enter details of the next sheep: " << endl;
    temp.input ();
    s.push_back (temp);

    cout << "Enter another? (y/n): ";
    cin >> answer;

    answer = tolower (answer);

    while (answer != 'y' && answer != 'n') {
      cout << "Please enter \'y\' or \'n\' ..." << endl;
      cout << "Enter another? (y/n): ";
      cin >> answer;
```

```
    }
  }

  return;
}

void readPigs (vector <Pig>& p)
{

  Pig temp;

  char answer;

  cout << "Enter details of the first pig: " << endl;
  temp.input ();
  p.push_back (temp);

  cout << "Enter another? (y/n): ";
  cin >> answer;

  answer = tolower (answer);

  while (answer != 'y' && answer != 'n') {
    cout << "Please enter \'y\' or \'n\' ..." << endl;
    cout << "Enter another? (y/n): ";
    cin >> answer;
  }

  while (answer == 'y') {
    cout << "Enter details of the next pig: " << endl;
    temp.input ();
    p.push_back (temp);

    cout << "Enter another? (y/n): ";
    cin >> answer;

    answer = tolower (answer);

    while (answer != 'y' && answer != 'n') {
      cout << "Please enter \'y\' or \'n\' ..." << endl;
      cout << "Enter another? (y/n): ";
      cin >> answer;
    }
  }

  return;
}

Pig mostValuablePig (vector <Pig> p)
{
  Pig bestPig = p[0];
  float bestValue = p[0].getValue ();

  for (int i = 0; i < p.size (); i ++) {
    if (p[i].getValue () > bestValue) {
      bestPig = p[i];
      bestValue = p[i].getValue ();
    }
  }
```

```
  return bestPig;
}

Sheep mostValuableSheep (vector <Sheep> s)
{
  Sheep bestSheep = s[0];
  float bestValue = s[0].getValue ();

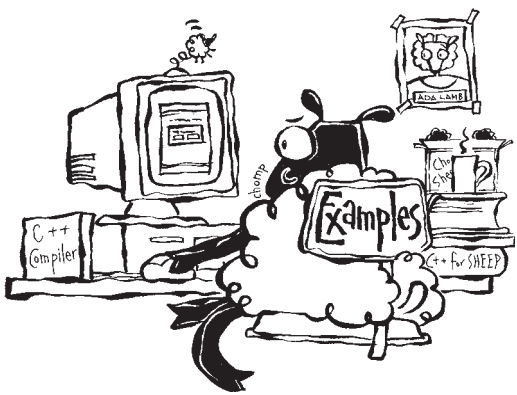
  for (int i = 0; i < s.size (); i ++) {
    if (s[i].getValue () > bestValue) {
      bestSheep = s[i];
      bestValue = s[i].getValue ();
    }
  }

  return bestSheep;
}

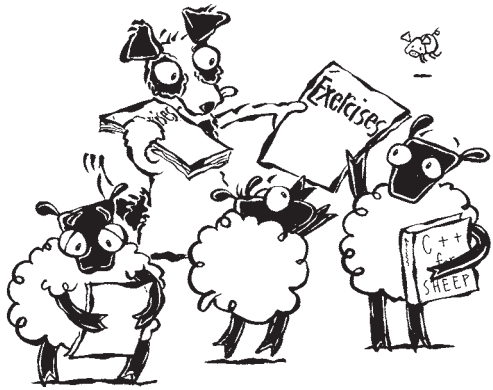
void mostValuable (vector <Pig> p, vector <Sheep> s)
{
  Pig topPig = mostValuablePig (p);
  Sheep topSheep = mostValuableSheep (s);

  cout << "The most valuable animal is: " << endl;
  if (topPig.getValue () > topSheep.getValue ()) {
    topPig.outputComplete ();
  }
  else {
    topSheep.outputComplete ();
  }

  return;
}
```



There are no examples this time; the programs in this chapter have provided quite enough. They have illustrated how a complete program is built up gradually by adding functions and how functions can often be used in many different programs.



18.1 The *Sheep* and *Pig* classes represented the coordinates of the animals as two separate integers. It would have been possible to implement a class "Point" to store coordinates and to provide suitable methods. An instance of this class could have been an attribute of both the animal classes. Would this have been a good design decision?

18.2 Adapt the program for sorting sheep based on their maximum possible value to sort pigs instead.

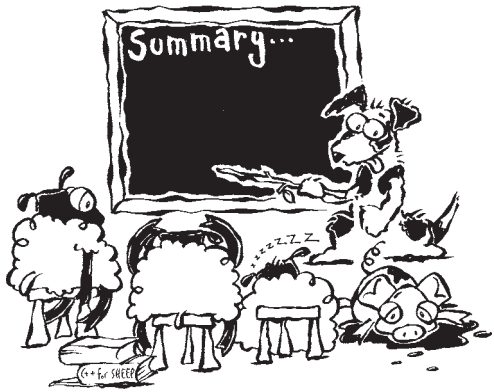
18.3 Write a program that reads the values for all the attributes of a flock of sheep and then displays the distances between each pair of sheep.

18.4 Sheep and pigs have been known to fight if they try to occupy the same point on the farm. Write a program that reads details of sheep and pigs and warns the farmer if any pairs are in danger of fighting. Two sheep may quite safely occupy the same point[6], as may two pigs.

18.5 Mollie wants a program that will keep track of the locations of two sheep. She wants to be able to enter the directions in which they move and have a display of their current positions. There should be a warning if the two sheep are at the same position or if either falls down the hole at (5, 3). After each pair of moves the program should ask Mollie if she needs to record another.

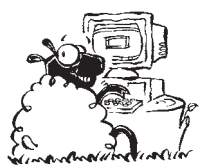
Write this program.

6 Yes. Quite.

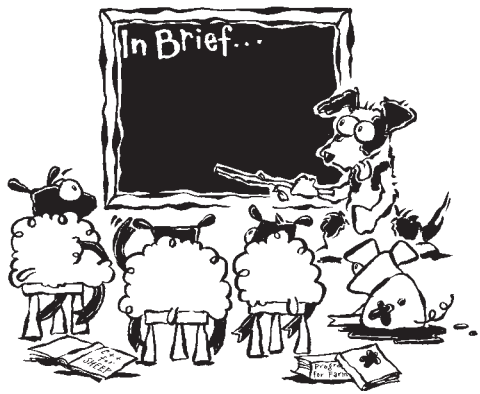


When developing a complex C++ program it is essential to develop the classes that the program will use first. This does not mean that the main function itself cannot be designed and perhaps even some of its functions started, but no serious work should be done until the classes are complete. The golden rule when developing a complex program is to develop and change one thing at a time; this way, if something goes wrong the cause can only be in one place.

Of course, complex programs need to be tested, as do classes. Before the end of the book the next chapter provides some final advice on how to go about testing programs.




Chapter 19
More
on
Testing



This is very nearly the end of the book. You have now seen all the C++ that you are going to see in detail. You should by now be able to write quite complex C++ programs making use of many of the features available in the current versions of C++. You should be able to analyse a problem and then go on to design and develop the classes and programs that will form the basis of a solution.

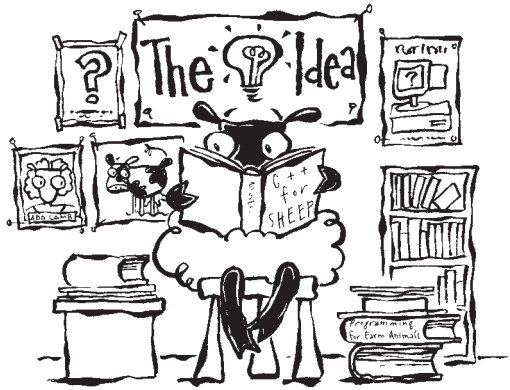
Of course, no solution is of the slightest use unless it works correctly. Many, many pages ago Chapter 9 explained the importance of testing and showed you how to write a *test plan* for each of your programs. Hopefully you've been doing this as you've written all your programs since then! There is truly no point whatsoever in spending time on developing a program unless you also take the time to test it thoroughly. An untested program is a worthless program and is potentially a very dangerous program.

Since Chapter 9, the programs you have been working on have become longer and more and more complicated. This chapter briefly explains how to test larger programs. The processes for testing larger programs are much the same as in Chapter 9, so you might want to go back and quickly read through it again. Larger programs bring with them the need for a very structured approach to testing; unstructured and haphazard testing can be worse than none at all.

Even with all this brave talk of testing it is important to remember that it is never possible to prove that a program works correctly in every case. Testing can only build up confidence in a program, it can never provide complete confidence. It is rarely, if ever, possible to test a program with all the input values that it will have to process. Nevertheless, the ideas in this chapter should mean that you are able to have a reasonably high level of confidence in your programs.

After reading this chapter you should be able to systematically test large programs and classes. You should be able to write a driver program for a class, and you should be able to use it to test a class. You should understand the difference between *black box* and *white box* testing, and you should understand how to decide which to use for a particular test. You should also be able to *dry run* a program.

Your programs should have a much better chance of working correctly!



Testing a program thoroughly is just as important as writing the program in the first place. In many ways it is more important, as an untested program may contain many costly errors. Since the last chapter on testing your programs have made use of more and more features of C++ and have become more and more complicated. It is crucial that you always test these complex programs very thoroughly.

This chapter is not intended to provide a complete description of the correct way to test programs. Whole books have been written on testing, and a complete description is way beyond the scope of any book on programming. Instead, this chapter gives you some ideas of ways in which you can test your programs and tries to convince you of the importance of testing.

You are already familiar with the idea of a *test plan* made up of a number of *test cases*. This idea translates directly to longer programs; the only thing that needs to be ensured is that the test cases in the plan cover every possible way in which a program can execute, or every possible route through a program. Your programs now include conditional statements and loops, so there are now many different ways in which they can execute; there must be enough test cases to test each of these possible execution routes thoroughly.

The test plans that you have written so far have relied on specifying a set of input values and the expected output values. The programs have been tested with each case, and if the actual output matches the expected output in each case the program has passed the tests. The approach is fine for simple and short programs, but it is rarely sufficient for more complex programs. A more thorough approach to testing is needed.

Your programs now also make use of classes, and these also have to be tested. A class is a building block that is potentially useful in many programs. Each class that a program makes use of must be properly tested before it is used; there is very little point in writing and testing a program that relies on a class that doesn't work properly. This chapter ends by explaining how to test a class before it is ever used in a program.

Routes through a program

By now your programs will probably be quite complex and there will be many possible routes through. Conditional statements and loops control the possible routes and each provides several possible ways for a program to execute. A test plan must make sure that every possible route through a program

is properly tested. This means first of all that every possibility from each conditional statement must be tested, as must every possible outcome of each loop.

When a user is presented with a prompt to enter a value into a program there is very little that limits what they can provide. Many failures in programs are caused by users managing to provide completely unforeseen input values and so managing to take completely unexpected routes through the program. It is absolutely vital that every possible route through a program is thoroughly tested, even if it seems very unlikely that it will ever be used.

First, each possibility in a conditional statement must be tested. For example, in a program including this conditional statement:

```
int aNumber;

if (aNumber > 1 && aNumber < 10) {
  // statements
}
else if (aNumber >= 10 && aNumber < 20) {
  // statements
}
else {
  // statements
}
```

the test plan must include values of *aNumber* that will cause each of the three possible sets of statements to be executed. These will very probably already be boundary values in the test plan, but more values must be included if not.

When testing a loop the crucial thing to consider is the initial value of the condition controlling the loop. Here:

```
int aNumber;
while (aNumber < 10) {
  // statements
}
```

it is important to test the program with values that will make the condition true and so cause the loop to execute, and also values that will make the condition false, and so mean that the loop is never executed.

It should be possible to represent each possible route through a program by a set of input values. If it is not possible to specify a set of values that will cause a certain route to be followed this raises the question of why the route is there! This list of values can be quite long, even for quite a small program:

```
// grades.cc
//
// Translates numeric grades to letters.
//
// Chapter 19.
//
// Author:    AMJ
// Date:      1st January 2002
// Platform:  g++, Linux

#include <iostream>

using namespace std;
```

```
int main ()
{

  int grade;

  char answer;

  do {

    cout << "Enter the grade: ";
    cin >> grade;

    if (grade > 69) {
      cout << grade << " is an A" << endl;
    }
    else if (grade > 59) {
      cout << grade << " is a B" << endl;
    }
    else if (grade > 49) {
      cout << grade << " is a C" << endl;
    }
    else if (grade > 39) {
      cout << grade << " is a D" << endl;
    }
    else {
      cout << grade << " is an E" << endl;
    }

    cout << "Run Again? (y/n) ";
    cin >> answer;

    while (answer != 'y' && answer != 'n') {
      cout << "Please enter \'y\' or \'n\'" << endl
           << "Run Again? (y/n) ";
      cin >> answer;
    }
  } while (answer == 'y');

  return 0;
}
```

This program (which I believe is correct!) converts a numeric grade into a simple letter scale. It also includes a simple dialogue to allow the user to run the program again. It is fairly trivial.

The test plan for this program is much less trivial. It must include:

- values of *grade* that cause each possibility in the conditional statement to be executed
- values of *answer* that cause each possible dialogue to be executed

The first of these is quite straightforward. Each possibility requires boundary values and at least one typical value. The dialogue is more complex. The testing for this must deal with the possibility that the user provides a correct value of 'y', a correct value of 'n', and a range of incorrect values. This must be tested in both places where the user is asked to enter a value, even though the code appears to be identical.

The three possible test cases for the first prompt in the dialogue are:

- the user enters 'y' to the first prompt, where the program should prompt for another grade
- the user enters 'n' to the first prompt, where the program should finish
- the user enters another value to the first prompt, where an error message should be displayed and the user prompted again

The same test cases also apply to the second prompt. It is easy to see how identifying all the possible cases can take some time. It is still important to do this thoroughly.

There may be a temptation to test parts of a program in isolation. In this example the `switch` statement and the two parts of the dialogue could be isolated and tested separately. This would shorten the test plan and the time taken for testing considerably, but it could lead to overconfidence in the program. In a program of any size it is entirely possible that a statement in one part of the program can have an unexpected side effect elsewhere in the program. While testing of the individual parts of a program is still a good idea, it is vital that the final program is tested as a complete unit before it is released.

Other ways of testing

So far, all the testing described has concentrated on providing input values and examining the output values. If the output values are as expected then the program is assumed to work for that particular test case. In this process the program itself has been treated as a "black box", which is precisely what this type of testing is called. The program is a black box in that its inner workings are never examined; if it produces the correct output then it is assumed to work.

Black box testing is usually effective, but thorough testing requires that the program itself be examined. To see why this can be necessary consider this example:

```
// add.cc
//
// Adds two integers.
//
// Chapter 19.
//
// Author:    AMJ
// Date:      1st January 2002
// Platform:  g++, Linux

#include <iostream>

using namespace std;

int main ()
{
  int first, second;

  cout << "Enter the first number: ";
  cin >> first;
  cout << "Enter the second number: ";
  cin >> second;

  cout << "The total is "
       << first + first << endl;
```

```
    return 0;
}
```

You can probably immediately see the error in this program. If you can't, look closely at the final `cout` line. The error is easy to see, but only because the program code is available. This program would pass some test cases in any black box testing; cases where the two numbers input were the same would appear to work. It is even possible that a poorly written black box test plan would lead someone to assume that the program did indeed work as expected.

Testing by examining the program is called "white box" testing. The program is now a white box and its statements are available to be examined. There are two possible approaches to white box testing. Both involve examining a printout of the program rather than running it on a computer.

The first approach is to "dry run" the program. This involves taking a test case and going through the program by hand, keeping track of the values of all the variables and of what is output. In a way this is the same as black box testing because the final output is available, but the advantage is that all the intermediate values are available too. This process can also be a very useful tactic when debugging a program rather than testing it.

The other approach is similar but involves two people. The idea is that someone, probably the programmer, "walks through" the program with someone else. The walkthrough involves explaining each line of the program in turn and explaining the possible routes through, much like a dry run. The process of working through the program can often highlight errors or reveal cases that have not been considered. The programmer is rarely the best person to test a program and should certainly not be allowed to test the program alone. The presence of another person makes the testing more likely to be thorough.

Black box or white box?

Both these approaches to testing have their advantages. White box testing can often find errors in programs that black box testing cannot and can sometimes show up holes in the black box test plan. A black box test can only ever show the presence of an error in the program; it can never show for certain exactly what it wrong. It is only when the program is examined in a white box test that the cause of the error can be found.

For thorough testing both approaches should be used. It is probably sensible to use black box testing first, not least because most of the time the process can be automated. But when a program passes all the black box tests it should still be thoroughly examined in a proper white box test.

Testing classes

All the discussion of testing so far has focused on testing programs. This has assumed that the classes used by the programs are known to work correctly. Obviously for this to be the case the classes themselves must also have been tested. They should certainly be thoroughly tested before they are ever used in a program. It can be very frustrating and time-consuming to find and correct an error in a program only to discover that the cause or the error is hidden away in a class used by the program.

Of course, classes cannot be tested in the same way as programs. Classes cannot be run; there is no input to provide and no output to trace. The key to testing a class is to use a small program that tests each method of the class in

turn. This allows for black box testing; the methods themselves should also be examined in a white box test. We have met such *driver programs* before, notably in the Case Study where one was written to test the *Sheep* class.

When a class is being developed a program to test it may already be provided or may be developed by another programmer. There may be a *test harness*: a program that can take the implementation of the class and test it. The class will be acceptable only when it passes all the tests run by the test harness successfully. Alternatively, the programmer who developed the class may have to write a simple driver program that will allow the class's methods to be tested.

A driver program does not need to be very sophisticated. At the simplest level it can simply call each method in turn and report the results. Here is the definition of yet another simple *Sheep* class:

```
// sheep.h
//
// Definition of a Sheep class. Keeps name and
//   details of location in field.
//
// Chapter 19.
//
// Author:    AMJ
// Date:      3rd January 2002
// Platform:  g++, Linux

#include <string>

using namespace std;

class Sheep {
  private:
    string name;
    int x;
    int y;

  public:
    Sheep ();

    // selectors

    string getName () const;
    int getX () const;
    int getY () const;

    // mutators

    void setName (string);
    void setX (int);
    void setY (int);

    // Move in compass directions

    void moveNorth ();
    void moveSouth ();
    void moveEast ();
    void moveWest ();
```

```
    // Move to specific point

    void moveTo (int, int);

    // True if at postition (x, y) in parameters

    bool atPosition (int, int) const;

    // Neatly formatted output in form (x, y)

    void printLocation () const;
};
```

The first and simplest approach to writing a driver is simply to write a program that calls each of the methods and displays the results. These are examined to see if they are as expected, in a way that is very similar to black box testing. A simple driver program for this class is:

```
// sheepDriver.cc
//
// Driver program for testing Sheep class.
//
// Chapter 19.
//
// Author:    AMJ
// Date:      3rd January 2002
// Platform:  g++, Linux

#include <iostream>
#include <string>

#include "sheep.h"

using namespace std;

int main ()
{

  Sheep s;

  // test mutators and selectors

  cout << "Setting name to \"Flossy\" ..." << endl;
  s.setName ("Flossy");
  cout << "Name is now \"" << s.getName ()
       << "\"" << endl;

  cout << "Setting position to 5, 3 ..." << endl;
  s.setX (5);
  s.setY (3);
  cout << "Position is now " << s.getX () << ", "
       << s.getY () << endl;

  // movement in directions

  cout << "Moving north ..." << endl;
  s.moveNorth ();
  cout << "Position is now " << s.getX () << ", "
       << s.getY () << endl;
```

```
    cout << "Moving south ..." << endl;
    s.moveSouth ();
    cout << "Position is now " << s.getX () << ", "
         << s.getY () << endl;

    cout << "Moving east ..." << endl;
    s.moveEast ();
    cout << "Position is now " << s.getX () << ", "
         << s.getY () << endl;

    cout << "Moving west ..." << endl;
    s.moveWest ();
    cout << "Position is now " << s.getX () << ", "
         << s.getY () << endl;

    // moveTo

    cout << "Moving to 0, 0 ..." << endl;
    s.moveTo (0, 0);
    cout << "Position is now " << s.getX () << ", "
         << s.getY () << endl;

    // atPosition

    cout << "Testing atPosition ..." << endl;
    if (s.atPosition (0, 0)) {
      cout << "atPosition is correct" << endl;
    }
    else {
      cout << "Problem with atPosition" << endl;
    }

    // neat output

    cout << "Testing formatted output ..." << endl;
    cout << "Position is now ";
    s.printLocation ();
    cout << endl;

    return 0;

  }
```

When it is run the program announces which methods it is using and displays the results:

```
Setting name to "Flossy" ...
Name is now "Flossy"
Setting position to 5, 3 ...
Position is now 5, 3
Moving north ...
Position is now 5, 4
Moving south ...
Position is now 5, 3
Moving east ...
Position is now 6, 3
Moving west ...
Position is now 5, 3
Moving to 0, 0 ...
```

```
Position is now 0, 0
Testing atPosition ...
atPosition is correct
Testing formatted output ...
Position is now (0, 0)
```

The programmer can determine the correct results by hand and then examine those produced by the driver program to see whether the class is working as expected.

A more thorough approach, and one that allows a complete test plan to be carried out, involves writing a simple menu to allow the user to call each method as required. This type of driver program can be quite long but it is easy to write; this is the approach that was used in the case study.

The menu provided by such a program can then be used to carry out the test plan.

```
// sheepDriver.cc
//
// Driver program for testing Sheep class.
//
// Chapter 19.
//
// Author:    AMJ
// Date:      3rd January 2002
// Platform:  g++, Linux

#include <iostream>
#include <string>

#include "sheep.h"

using namespace std;

char menu ();

int main ()
{

  Sheep s;

  char choice;

  string name;
  int x, y;

  // display menu

  do {
    choice = menu ();

    switch (choice) {
      case '1': cout << "Enter New Name: ";
                cin >> name;
                s.setName (name);
                break;
      case '2': cout << "Enter New X: ";
                cin >> x;
                s.setX (x);
```

```
                break;
        case '3': cout << "Enter New Y: ";
                cin >> y;
                s.setX (y);
                break;
        case '4': cout << "Name is \"" << s.getName ()
                     << "\"" << endl;
                break;
        case '5': cout << "X is " << s.getX () << endl;
                break;
        case '6': cout << "Y is " << s.getY () << endl;
                break;
        case 'N': s.moveNorth ();
                break;
        case 'S': s.moveSouth ();
                break;
        case 'E': s.moveEast ();
                break;
        case 'W': s.moveWest ();
                break;
        case 'M': cout << "Enter New X: ";
                cin >> x;
                cout << "Enter New Y: ";
                cin >> y;
                s.moveTo (x, y);
                break;
        case 'A': cout << "Enter X to Check: ";
                cin >> x;
                cout << "Enter Y to Check: ";
                cin >> y;
                if (s.atPosition (x, y)) {
                  cout << "Sheep is there";
                }
                else {
                  cout << "Sheep is not there";
                }
                break;
        case 'P': cout << "Location is ";
                s.printLocation ();
                cout << endl;
                break;
      } // switch
    } while (choice != 'Q');

    return 0;

}

char menu ()
{

    char choice;

    cout << "Sheep Driver" << endl
         << "~~~~~~~~~~~~" << endl << endl
         << "1. Set Name" << endl
         << "2. Set X" << endl
         << "3. Set Y" << endl
         << "4. Get Name" << endl
```

```
        << "5. Get X" << endl
        << "6. Get Y" << endl
        << "N. Move North" << endl
        << "S. Move South" << endl
        << "E. Move East" << endl
        << "W. Move West" << endl
        << "M. Move to X, Y" << endl
        << "A. Check Position" << endl
        << "P. Print Location" << endl << endl
        << "Q. Quit" << endl << endl;

    cout << "Choice: ";
    cin >> choice;

    return choice;
}
```

As before, the programmer can work out a set of tests and can then apply them using the menu. The actual results can be compared to those produced to determine whether the class works correctly[1].

These driver programs are simply used to test the programs, so there is no need for any sophisticated error checking in them, or even for any checking of the user's input. They simply allow the tester to use the class and to examine the results. When armed with a suitable plan the tester should be able to discover whether or not the class works as expected. There is an implicit assumption here, of course: that the driver programs are correct. They should also be tested! It is very annoying to discover that an apparent error in a class is caused by sloppy programming in the driver.

The implementation of the *Sheep* class that is being tested here is on the web site.

Building and testing a program

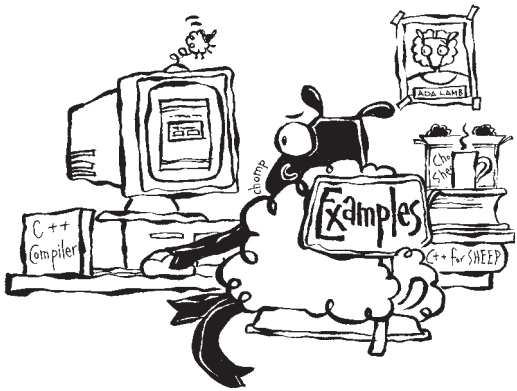
A complete C++ program involves classes as well as the program itself. It is important that the classes are written and tested before the program is started (or at least before there is any contact between the two). There is very little point in writing a program that relies on an untested class; if the two are developed and tested separately it is always much easier to trace and correct errors and bugs. This does not eliminate the possibility that writing a program using a class will show up errors in the class, but it should reduce the likelihood.

Thorough testing is a vital part of developing a program. It is even more important in a language like C++, where programs are built up of various components and are often written in many separate files. You should always test your programs and you should certainly never be tempted to regard testing as an afterthought.

Never be tempted to say things like "It looks as if it works"; always be sure that it does work!

---

1 I almost wrote this as "appears to work correctly". Remember that this is what testing is demonstrating. Testing gives us confidence that a program or class works as we expect and want; testing can never prove that a program or class works correctly.



Example 1 – Counting letters

Flossy has decided to take up word games. To help her in this new enterprise she has written a simple program to count the number of vowels and consonants in a word. Unfortunately, her program is not producing the correct results. Flossy cannot see the error, so she decides to dry run the program to see what that reveals.

Here is the program:

```
// wordStats.cc
//
// Counts vowels and consonants in a word.
//
// Chapter 19, Examples 1.
//
// Author:    AMJ
// Date:      3rd January 2002
// Platform:  g++, Linux

#include <iostream>
#include <string>

using namespace std;

int main ()
{

  string word;

  int consonants = 0;
  int vowels = 0;

  // get word

  cout << "Enter word: ";
  cin >> word;

  // analyse

  for (int i = 0; i < word.length () - 1; i ++) {
    if (word.at (i) == 'a' || word.at (i) == 'e'
        || word.at (i) == 'i' || word.at (i) == 'o'
```

```
          || word.at (i) == 'u') {
        vowels ++;
      }
      else {
        consonants ++;
      }
    }

    // output results

    cout << "\"" << word << "\" contains ";

    if (vowels == 1) {
      cout << "1 vowel and ";
    }
    else {
      cout << vowels << " vowels and ";
    }

    if (consonants == 1) {
      cout << "1 consonant" << endl;
    }
    else {
      cout << consonants << " consonants" << endl;
    }

    return 0;

  }
```

What would the dry run reveal? What is the error?

There is nothing obviously wrong with this program. Running it would reveal that the results are always wrong. You might want to get hold of the program from the web site and try it out. Remember that it doesn't work!

Examples of its output are:

```
Enter word: sheep
"sheep" contains 2 vowels and 2 consonants
Enter word: flossy
"flossy" contains 1 vowel and 4 consonants
```

Some people (people who are very good at word games and puzzles, probably) might be able to deduce what the error is from just these two examples, but dry running the program provides a more structured way to investigate what is going on.

The first stage is to construct a simple table to keep track of the values of the variables. The initial values of the variables are added first:

	word	consonants	vowels	i
Initial		0	0	

Then the program is examined a line at a time, and each change in the value of a variable is recorded in the table. Immediately before the `for` loop is reached the table reads:

	word	consonants	vowels	i
Initial		0	0	
Before loop	sheep	0	0	

The loop uses two methods of the string class, `at` and `length`. To keep track of these it is easiest to add them to the table too:

	word	consonants	vowels	i	word.length	word.at(i)
Initial		0	0			
Before loop	sheep	0	0			

The table is updated again and again as the program progresses. By the end of the first execution of the loop it is:

	word	consonants	vowels	i	word.length	word.at(i)
Initial		0	0			
Before loop	sheep	0	0			
After loop 1	sheep	1	0	0	5	s

After this point the condition controlling the loop:

```
i < word.length () - 1
```

can be seen from the table to correspond to:

```
0 < 5 - 1
```

which is plainly still true, so the loop will execute again. This continues:

	word	consonants	vowels	i	word.length	word.at(i)
Initial		0	0			
Before loop	sheep	0	0			
After loop 1	sheep	1	0	1	5	s
After loop 2	sheep	2	0	2	5	h
After loop 3	sheep	2	1	3	5	e
After loop 4	sheep	2	2	4	5	e

At this point the condition is still:

```
i < word.length () - 1
```

which the table reveals is now:

```
5 < 5 - 1
```

or indeed:

```
5 < 4
```

which is obviously false. This means that the loop will stop and the results will be output.

It should now be clear what the error is; the final character of the string is never being processed. The mistake is in the `for` loop continuation condition:

```
for (int i = 0; i < word.length () - 1; i ++)
```

which is clearly causing the loop to execute one too few times. The solution is obvious:

```
for (int i = 0; i < word.length (); i ++)
```

This change will[2] make the program work correctly.

Example 2 – Flossy's averages again

Mollie has recently given the sheep a programming test and Flossy has been working on a simple program to work out the average marks they scored. She has tested the program carefully, but is confused that it seems to work sometimes but not all the time. She is especially baffled that her black box test cases seem to give different results every time she runs her program.

What should she do now? What is wrong with her program?

Problems such as this are often very difficult for the programmer to find. Black box testing is clearly not helping very much, so some sort of white box testing is the only way forward. It is quite likely that by showing her program to another programmer Flossy could quickly have the program solved.

Here is Flossy's program:

```
// averages.cc
//
// Calculates average marks on a test.
//
// Chapter 19, Examples 2.
//
// Author:    AMJ
// Date:      3rd January 2002
// Platform:  g++, Linux

#include <iostream>

using namespace std;

int main ()
{
  int numGrades;
  int grade;
  int total;

  // get number of grades

  cout << "Enter number of grades: ";
  cin >> numGrades;

  while (numGrades < 1) {
    cout << "Error. Must be at least 1. ";
    cout << "Enter number of grades: ";
    cin >> numGrades;
  }

  // get grades and calculate total

  for (int i = 0; i < numGrades; i ++) {
```

2 Should?

```
    cout << "Enter grade #" << i + 1 << ": ";
    cin >> grade;

    while (grade < 0) {
      cout << "Error. Must be at least 0. ";
      cout << "Enter grade #" << i + 1 << ": ";
      cin >> grade;
    }
    total += grade;
  }

  // display average

  cout << "Average grade is "
       << float (total) / numGrades << endl;

  return 0;

}
```

The error in the program is in fact very subtle. If you can see it immediately, go and get yourself a prize!

It is precisely the sort of error that another programmer seeing this program for the first time could well spot immediately, but which the original programmer could spend many fruitless hours trying to find.

Flossy has forgotten to initialise the *total* variable. This means that it initially contains whatever value happened to be at the memory location allocated to it. Most of the time this will probably be 0, in which case the program would work correctly, but some of the time it will be something else, and this will introduce the error.

This explains why the program would appear to work some of the time but not all the time. The behaviour of the program depends on the allocation of the memory and what is found there; this is basically random, so the program appears to behave randomly.

Subtle errors such as this are often best found by other programmers.



19.1 Examine this conditional statement. What is odd about it?

```
int aNumber;

if (aNumber < 10) {
  // statements
}
else if (aNumber >= 10} {
  // statements
}
else {
  // statements
}
```

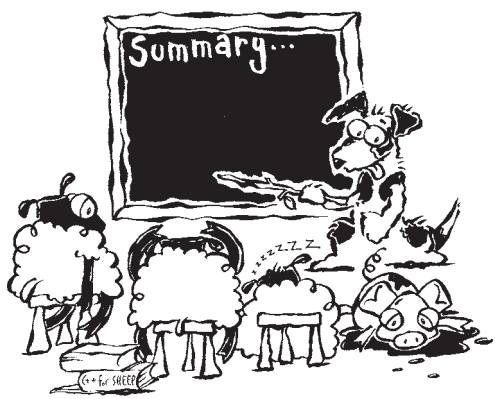
19.2 A programmer shows you a program that is not working correctly. You see this loop in the program. What should you immediately suspect?

```
while (counter = 1) {
  // statements
  counter ++;
}
```

19.3 Write a program that prompts the user to enter five words and then sorts them into alphabetical order. Use a vector and a Bubblesort. Dry run the sorting function in your program with the words "sausage", "egg", "bacon", "tomato" and "mushroom".

19.4 Implement the *Sheep* class from this chapter. Use a driver program to test your implementation.

19.5 Take the driver program for the *Sheep* class on pages 320–322. Use it to test the methods in the class. Pay particular attention to the methods that were not used in any of the programs!



As programs become more and more complex, so does the process of testing. Testing should never be ignored. It should be seen as part of the process of programming; a good programmer will always test the program as it is being developed. A program is not complete when it has been written; it is complete only when it has been thoroughly tested.

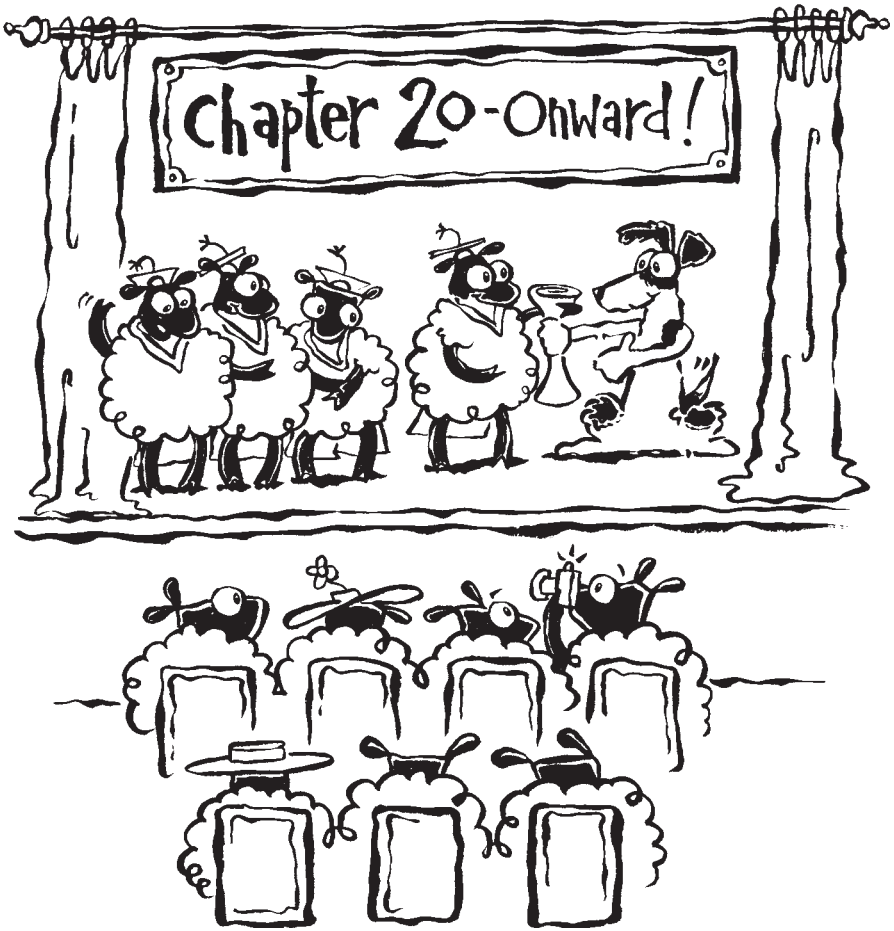
When you start to learn to program there can be a temptation to be so relieved that a program works that you cannot bear to test it. Testing can uncover mistakes that you have made and can mean that you are in for a lot more work. Try not to succumb to this temptation. Always test your

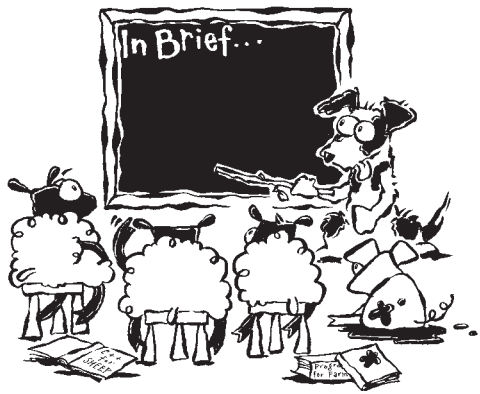
programs as thoroughly as you can, and never be afraid to help other people test theirs.

Testing should always be a structured process and should always be carried out according to a plan. The plan should be written before the program; don't be tempted to write the test plan to fit the parts of the program that work and to hide the rest!

Testing a program is as important as developing it in the first place. Without thorough testing any program is worthless.




Chapter 20-Onward!



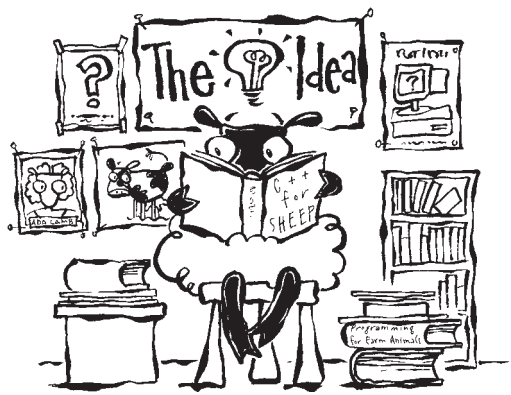
This is the end of the book, but it is not the end of C++.

Far from it, in fact.

C++ has a great many more features that you haven't yet seen or had any need to use. C++ is continuing to develop and more and more features will certainly be added in the future. As C++ becomes more popular, and as computer programs become more and more complex, more and more powerful features are added to the language. Although you haven't seen any of these yet you should now be able to pick them up as and when you need to use them.

This chapter describes five of the more advanced features that would make a good next step in your learning. There's nothing especially complicated here, but all this chapter aims to give you is a quick overview. By now you should be able to understand a new programming concept and then go and try it out for yourself.

After reading this chapter you should know about some of the more advanced features of C++ that have not been covered in this book. You should understand what they offer and you should know how to find out more about them if you need to.



This chapter describes five more advanced features of C++:

- input and output using files

- command line arguments
- inheritance
- formatted output
- operator and function overloading

Each of these is described in turn; they are arranged in no particular order.

These sections do not aim to provide a complete description of these ideas. They should give you a flavour of what each topic is about; this should be just about enough so that you can try them out yourself in your own programs.

Input and output using files

All of the useful programs that you have written have involved a user in entering some values for the program to process. The results of this processing have later been displayed on the user's screen. You have only been able to get input from the keyboard and to send output to the screen.

This is often far too restrictive. It would be much more useful if the results from running a program could be stored in a file on the computer so that they could later, for example, be used as input to another program. In many of the examples in this book the user has been asked to enter a great deal of data over and over again; it would surely be better if this data could be stored in a file.

Input from the keyboard is achieved using `cin >>` and output is similarly achieved using `cout <<`. These are the default input and output streams. It is possible to define other streams that correspond to files on the computer. These streams can then be used for input and output, just like the keyboard and screen.

Using files for input or output requires that the `fstream` library is included:

```
#include <fstream>
```

The streams are declared in the same way as variables. A stream that is to be used for input has the type `ifstream` and one for output has the type `ofstream`. When each is declared it is given an identifier in the same way as any other variable. This is not to be confused with the name of the file on the computer.

```
ifstream inFile;    // input stream
ofstream outFile;   // output stream
```

The name of the file on the computer that is associated with each stream can be specified at the same time as the declaration:

```
ifstream inFile ("data.dat");
ofstream outFile ("output.txt");
```

These streams can then be used in exactly the same way as `cin` and `cout`. The operators `>>` and `<<` work as would be expected for input and output respectively.

```
int aNumber;

ifstream inFile ("data.dat");
ofstream outFile ("output.txt");

inFile >> aNumber;          // reads number from data.dat
outFile << aNumber << endl; // writes to output.txt
```

Streams are, unsurprisingly, classes and there are many methods that can be used with them. The most common ones are:

- `open` – opens a file for input or output
- `close` – closes a file when input or output have finished
- `eof` – returns true if the end of the file has been reached
- `fail` – returns true if any attempt to read from or write to a file was not successful

Where the names of the files to be used are not known in advance the user can be asked to enter the names and the `open` method can be used to associate the stream with the file. In this case a small complication arises because the parameter of the `open` method must be a C "string" and not a C++ string. This means that this is an error:

```
ifstream inFile;
string filename;

cout << "Enter the file name: ";
cin >> filename;

inFile.open (filename);           // Error!
```

The solution is quite simple, but rather messy. The string class has another method, `c_str`, that converts a C++ string into the C equivalent. So the line:

```
inFile.open (filename.c_str ());  // Correct!
```

will open the file correctly.

To show how output streams can be used to output to a file rather than to the screen here is a version of Flossy's batting average program that outputs to a file:

```
// sortSheep.cc
//
// Sorts a team of sheep into batting average order.
//
// Chapter 20.
//
// Author:    AMJ
// Date:      2nd January 2002
// Platform:  g++, Linux

#include <iostream>
#include <fstream>
#include <string>

using namespace std;

void sortTeam (string[], float[], int);

int main ()
{

  const int SheepInTeam = 11;

  string names[SheepInTeam];
  float averages[SheepInTeam];
```

```
    // open a file to store the output

    ofstream outFile ("batting.txt");

    // get names and averages

    for (int i = 0; i < SheepInTeam; i ++) {
      cout << "Enter Name of Sheep #" << i + 1 << ": ";
      cin >> names[i];

      cout << "Enter " << names[i] << "\'s average: ";
      cin >> averages[i];

      while (averages[i] < 0.0) {
        cout << "Error! Average cannot be negative."
             << endl;
        cout << "Enter " << names[i] << "\'s average: ";
        cin >> averages[i];
      }
    }

    // sort

    sortTeam (names, averages, SheepInTeam);

    // output sheep to file

    for (int i = 0; i < SheepInTeam; i ++) {
      outFile << names[i] << ": " << averages[i] << endl;
    }

    cout << "Sorted list written to \"batting.txt\""
         << endl;

    return 0;
  }

  void sortTeam (string n[], float a[], int num)
  {
    string tempName;
    float tempAverage;

    for (int p = 0; p < num - 1; p ++) {
      for (int e = 0; e < num - 1; e ++) {
        if (a[e] < a[e + 1]) {
          tempName = n[e];
          tempAverage = a[e];
          n[e] = n[e + 1];
          a[e] = a[e + 1];
          n[e + 1] = tempName;
          a[e + 1] = tempAverage;
        } // if
      } // each pass
    } // total passes

    return;

  }
```

This program assumes that the file can be opened (effectively it assumes that a file of that name exists), which is probably not always the case. A simple check can be added to ensure that an opened file can be accessed:

```
ifstream inFile ("data.dat");
if (!inFile) {
  cerr << "Error: file not found" << endl;
  exit (1);
}
```

This tests whether the file was opened successfully, and prints an error message if not. It is normally a very serious error if a program is unable to open a file that is needed for input, so it is usual for the program to terminate immediately, as it does here.

As a final example of using files, here is a program that counts the number of words in the file. The comments explain in more detail how it works.

```
// wordCount.cc
//
// Counts words in a file.
//
// Chapter 20.
//
// Author:    AMJ
// Date:      3rd January 2002
// Platform:  g++, Linux

#include <cstdlib>
#include <fstream>
#include <iostream>
#include <string>

using namespace std;

int main ()
{

  ifstream inFile;
  string filename;

  int words = 0;
  string temp;

  // get file name

  cout << "Enter the file name: ";
  cin >> filename;

  // open file and check it exists

  inFile.open (filename.c_str());

  if (!inFile) { // file does not exist
    cerr << "Error: File cannot be opened" << endl;
    exit (1);
  }

  // read each word and count
```

```
    while (!inFile.eof ()) {  // loop until end of file
      inFile >> temp;
      if (!inFile.fail ()) {  // if input worked
        words ++;             // increment word count
      }
    }

    // display results

    cout << "The file contained ";

    if (words == 1) {
      cout << "1 word" << endl;
    }
    else {
      cout << words << " words" << endl;
    }

    return 0;

  }
```

The mechanism used in this program for reading the name of the file to be processed is rather cumbersome for both the programmer and the user. It would be much neater if the user could simply provide the name of the file at the same time as they start the program; on a Unix system or similar they would provide the name on the command line. C++ provides an easy way to let users do just this.

Command line arguments

Command line arguments in C++ are a feature inherited from C, but they are sufficiently useful to deserve a quick summary here. The idea is that the user provides some input values at the same time as they start the program; these values might be the name of the file to be processed, for example.

This idea will be very familiar to you if you have used a Unix or similar system. Unix commands are typed at a prompt and are often followed with a set of arguments. This is less familiar in the Windows world, but the idea is still there. Clicking on an icon to start a program is a command; clicking on a file to start a program associated with that file provides a command and an argument.

If command line arguments are to be used in a C++ program the first line of the main function must be changed to:

```
int main (int argc, char **argv)
```

The line must be exactly as shown. This declares two new variables, `argc` and `argv`. `argc` is the number of arguments that were found on the command line with the command itself counting as one. `argv` is an array of C strings that contain the arguments themselves.

For example, if the command line is:

```
prog file1 file2
```

where *prog* is a command and *file1* and *file2* are its two arguments. Here `argc` would have the value 3, since there are two arguments and the command

itself. argv would be an array of three "strings", namely "prog", "file1" and "file2".

The elements of argv can then be used in the program just as any other variable. The most common use of argc is to check that the expected number of arguments was actually provided. If a program expected exactly one argument on the command line the check would be:

```
if (argc != 2) { // argc includes the command
  cerr << "Error: wrong number of arguments" << endl;
  exit (1);
}
```

The wrong number of arguments is normally a fatal error and so the program exits immediately. In this example the error message is written to cerr, the standard error stream.

As an example of the use of command line arguments, here is a version of the program from the last section modified to use them rather than prompt the user to enter the file name.

```
// wordCount.cc
//
// Counts words in a file.
//
// Chapter 20.
//
// Author:    AMJ
// Date:      3rd January 2002
// Platform:  g++, Linux

#include <cstdlib>
#include <fstream>
#include <iostream>
#include <string>

using namespace std;

int main (int argc, char **argv)
{

  ifstream inFile;
  string filename;

  int words = 0;
  string temp;

  // check command line arguments

  if (argc != 2) {
    cerr << "Usage: " << argv[0] << " <filename>"
         << endl;
    exit (1);
  }

  // open file and check

  inFile.open (argv[1]);

  if (!inFile) {
```

```
    cerr << "Error: File cannot be opened" << endl;
    exit (2);
  }

  // read each word and count

  while (!inFile.eof ()) {
    inFile >> temp;
    if (!inFile.fail ()) {
      words ++;
    }
  }

  // display results

  cout << "The file contained ";

  if (words == 1) {
    cout << "1 word" << endl;
  }
  else {
    cout << words << " words" << endl;
  }

  return 0;

}
```

A handy side effect of the modification is that the string containing the name of the file is already in the required C format, so no conversion is required.

Inheritance

It is not uncommon for an application to contain many classes that are very similar. Applications will deal with different types of people, bank accounts, books or animals. There are many occasions when a method written for one class can be used in another and where different classes have the same attributes. In this case there can often be a great deal of code that has to be duplicated in the different classes. We have seen this several times when code for different types of animal has had to be duplicated in different classes.

Sometimes the classes form a hierarchy. If an application processes classes representing different kinds of people – students, teachers, bus conductors – many of the different classes will have the same attributes. It would be very wasteful to have to implement the same methods in each class.

C++ provides a mechanism to allow for this, called inheritance. Inheritance allows the programmer to define one basic class and to allow other classes to inherit all the attributes and methods of this class. The inheriting classes have all the attributes and methods of this class and can also have their own. The most common example is probably a class for *People* that would hold names, birthdays and so on and would be inherited by classes representing particular types of people such as students and bus conductors.

The class that contains the common attributes and methods is called the *base class* and the classes inheriting from it are called *derived classes*. The base class is defined first in the usual way:

```
#include <string>

using namespace std;

class Person {
  private:
    string name;
    int age;
    string address;

  public:
    Person ();

    void setName (string);
    void setAge (int);
    void setAddress (string);

    string getName () const;
    int getAge () const;
    string getAddress () const;
};
```

Derived classes include only the attributes and methods that are unique to them. All the other attributes and methods are inherited from the base class, the name of which is specified on the first line of the class's definition:

```
class BusConductor : public Person {
  private:
    int busRoute;

public:
  BusConductor ();
  void setBusRoute (int);
  int getBusRoute () const;
};
```

Inheritance can save a great deal of coding. There can be many levels of inheritance in an application with derived classes acting as base classes for others.

Formatted output

None of the programs so far in this book have attempted to have any sophisticated control over the precise format of the data that is output. The rule has basically been that if the results are right it doesn't much matter what they look like. This is obviously not an approach that should be adopted with more complicated programs. Much of the output produced by our programs so far has probably been a bit on the ugly side.

Unfortunately the facilities for formatting output are thought by many to be one of the serious weaknesses of C++. They are very complicated and more than a little cumbersome to use. Many programmers in fact prefer instead to use the facilities provided by C (which are also available in C++, of course).

Nevertheless, here are some examples of the facilities available. All of these require the `iomanip` library:

```
#include <iomanip>
```

The format of the output is controlled in two ways. The first is by adding directives to the output statement; these can control such things as the number of places of decimals printed for a floating-point value or the width of the field in which a value is displayed. The other technique involves setting various "flags" which control the ways in which values are displayed; this can be used, for example, to determine whether a decimal point is always displayed in a floating-point value and where within the field a value is justified.

If a program is displaying a set of floating-point values it is usual to ensure that all the values are displayed with the same number of decimal places. To set the precision (number of decimal places) used to output a floating-point number the directive `setprecision` is used:

```
const float PI = 3.1415;

cout << setprecision (2)
     << PI << endl;
```

This will output:

```
3.14
```

If a floating-point number has fewer than two significant places of decimals these are not printed by default, so:

```
float number = 1.0;

cout << setprecision (2) << number << endl;
```

would display:

```
1
```

If these extra places are required, a flag can be set:

```
float one = 1.0;

cout << setiosflags (ios::fixed)
     << setprecision (2)
     << one << endl;
```

which will output:

```
1.00
```

rather than the default:

```
1
```

The width of the field in which a value is printed can be set using the `setw` directive. This can be useful for creating tables of data with columns neatly lined up.

```
string flossy = "Flossy";

cout << flossy << endl;   // prints "Flossy"
cout << setw (20)         // sets field width to 20
     << flossy << endl;   // prints with trailing spaces
```

The values are normally aligned in the left of the field with spaces added to the end to make up the required column width.

In some tables numeric values are required to be aligned to the right of the field, and this is in fact the default. This default can be changed by setting

another flag. This statement will display the value right-justified in a field 10 characters wide:

```
const int Hundred = 100;

cout << setw (10) << Hundred << endl;
```

To change to left justification a flag must be set:

```
cout << setiosflags (ios::left)
     << setw (10) << Hundred << endl;
```

and to change back to right justification the flag is unset:

```
cout << resetiosflags (ios::left)
     << setw (10) << Hundred << endl;
```

The following program and its output illustrate these basic facilities. While there are many more available these should be enough for most programs.

```
// output.cc
//
// Demonstrates some formatted output.
//
// Chapter 20.
//
// Author:    AMJ
// Date:      2nd January 2002
// Platform:  g++, Linux

#include <iomanip>
#include <iostream>
#include <string>

using namespace std;

int main ()
{

  const float PI = 3.1415;
  const float One = 1.0;
  const string Flossy = "Flossy";
  const int Hundred = 100;

// precision

cout << "Precision can be altered using setprecision:"
     << endl;

cout << setiosflags (ios::fixed)
     << setprecision (2)
     << "PI to two decimal places: "
     << PI << endl
     << setprecision (3)
     << "and to three decimal places: "
     << PI << endl;

// if the decimal part is zero it is not printed
// unless setiosflags has been used

cout << endl
```

```
      << "Decimal places can be added with ios::fixed:"
      << endl;

  cout << resetiosflags (ios::fixed)
       << setprecision (2)
       << "Without ios::fixed: "
       << One << endl
       << setiosflags (ios::fixed)
       << "With ios::fixed: "
       << One << endl;

  // field width

  cout << endl
       << "setw can be used to set the field width:"
       << endl;

  cout << "Default:           "
       << "**" << Flossy << "**" << endl
       << "Field width of 20: "
       << "**" << setw (20) << Flossy << "**" << endl
       << "Field width of 10: "
       << "**" << setw (10) << Flossy << "**" << endl;

  // justification

  cout << endl
       << "setw and iosflags can control justification:"
       << endl;

  cout << "Default:                      "
       << "**" << Hundred << "**" << endl
       << "Default with field width 10: "
       << "**" << setw (10) << Hundred << "**" << endl
       << setiosflags (ios::left)
       << "The same left justified:     "
       << "**" << setw (10) << Hundred << "**" << endl
       << resetiosflags (ios::left)
       << "With the default restored:   "
       << "**" << setw (10) << Hundred << "**" << endl;

    return 0;
  }
```

This program outputs:

```
Precision can be altered using setprecision:
PI to two decimal places: 3.14
and to three decimal places: 3.141

Decimal places can be added with ios::fixed:
Without ios::fixed: 1
With ios::fixed: 1.00

setw can be used to set the field width:
Default:           **Flossy**
Field width of 20: **Flossy                  **
Field width of 10: **Flossy        **
```

```
setw and iosflags can control justification:
Default:                    **100**
Default with field width 10: **       100**
The same left justified:    **100       **
With the default restored:  **       100**
```

If nothing else this brief description should have shown you why some programmers prefer to use the formatted output of C. In any case the few techniques illustrated here should be enough for most simple programs.

Overloaded operators and functions

Functions in C++ are identified by their prototypes. The prototype states the name of the function, the type of the returned value, and the types of any parameters. It is this combination of information together that identifies the function; there is no need for every function in a program to have a different name.

The chapter on vectors and arrays included two prototypes for functions that sorted collections of values. One was for sorting a vector and the other was for sorting an array:

```
void sortVector (vector <int>&);
void sortArray (int[], int);
```

These functions have been given different names, but there is no particularly good reason to do this. If both functions were called simply *sort*:

```
void sort (vector <int>&);
void sort (int[], int);
```

it would still be obvious which one was being called from the type and number of parameters. This call:

```
sort (numbers);
```

is clearly calling the vector version, since only one parameter is provided, while this:

```
sort (numbers, count);
```

must be a call to the second version for arrays, as there are two parameters.

In this example the function name *sort* has been overloaded; there are two functions with this name. This is fine as long as the function that is being called can be uniquely determined by examining the type and number of its parameters.

This technique can make programs much more readable and can save the programmer from having to use what amount to artificial names for functions. For example, a function to return the highest integer in an array would have the prototype:

```
int highest (const int[], int);
```

If the same program also required a function to return the highest integer from three supplied as parameters this can have a prototype with the same function name because the parameters are different:

```
int highest (int, int, int);
```

This is far preferable to having to rename the functions to ensure that the names were unique:

```
int highestInArray (const int[], int);
int highestOfThree (int, int, int);
```

The following program illustrates this idea with some more examples. Each time the function *highest* is called the parameters uniquely determine which is to be used.

```
// overload.cc
//
// Demonstrates overloaded functions.
//
// Chapter 20.
//
// Author:    AMJ
// Date:      4th January 2002
// Platform:  g++, Linux

#include <iostream>

using namespace std;

int highest (int, int);
int highest (int, int, int);
int highest (const int[], int);
float highest (float, float);

int main ()
{

  const int One = 1;
  const int Two = 2;
  const int Three = 3;

  const int Numbers[4] = {1, 2, 3, 4};
  const float Four = 4.0;
  const float Five = 5.0;

  // two integer parameters

  cout << highest (One, Two) << endl;

  // three integer parameters

  cout << highest (One, Two, Three) << endl;

  // integer arrray

  cout << highest (Numbers, 4) << endl;

  // floating-point parameters

  cout << highest (Four, Five) << endl;

  return 0;

}
```

```
int highest (int x, int y)
{
  if (x > y) {
    return x;
  }
  else {
    return y;
  }
}

int highest (int x, int y, int z)
{
  if (x > y && x > z) {
    return x;
  }
  else if (y > z) {
    return y;
  }
  else {
    return z;
  }
}

int highest (const int n[], int c)
{
  int highest = n[0];
  for (int i = 1; i < c; i ++) {
    if (n[i] > highest) {
      highest = n[i];
    }
  }
  return highest;
}

float highest (float x, float y)
{
  if (x > y) {
    return x;
  }
  else {
    return y;
  }
}
```

Operators can also be overloaded. This is in fact already familiar from some of the operations on the basic data types. In this program:

```
string stringOne, stringTwo;
int intOne, intTwo;

if (stringOne < stringTwo && intOne < intTwo) {
  // statements
}
```

it is obvious that the two uses of the < symbol are different. While they both have the same logical meaning ("less than"), the first is comparing two strings while the second is comparing two integers. Behind the scenes these

two are very different. The first is applying what amounts to alphabetical order[1], while the second is applying number order.

The existing operators can be overloaded so that they will work with classes. As with any other method they are defined and implemented and can then be used in programs that make use of the class. A simple example adds a < operator to the definition of a simple *Sheep* class:

```
// sheep.h
//
// Definition of a Sheep class. Stores name, age.
// Includes an overloaded < operator.
//
// Chapter 20.
//
// Author:    AMJ
// Date:      2nd January 2002
// Platform:  g++, Linux

#include <string>

using namespace std;

class Sheep {
  private:
    string name;
    int age;

  public:
    Sheep ();

    string getName () const;
    int getAge () const;
    void setName (string);
    void setAge (int);

    void print () const;

    // overloaded < operator

    bool operator < (Sheep) const;
};
```

The < operator clearly returns a Boolean value and requires two instances of the class, the second provided as a parameter. In this simple example it does nothing more than compare the ages of the sheep and returns `true` if the first sheep is younger. It is implemented as usual in the class's implementation file:

```
bool Sheep::operator < (Sheep s) const
{
  return age < s.age;
}
```

where it makes use of the integer version of the < operator!

---

1 What is the order it uses exactly? *apples* is less than *pears*, but is *apples* less than *!!pears* or *Pears*? Find out!

The following program makes use of this operator. It is used in exactly the same way as the other versions of it.

```
// sheepOp.cc
//
// Demostrates overloaded < operator in the Sheep class.
//
// Chapter 20.
//
// Author:    AMJ
// Date:      4th January 2002
// Platform:  g++, Linux

#include <iostream>
#include <string>

#include "sheep.h"

using namespace std;

int main ()
{
  Sheep aSheep, anotherSheep;

  string tempName;
  int tempAge;

  // get names and ages

  cout << "Enter name of first sheep: ";
  cin >> tempName;
  cout << "Enter " << tempName << "\'s age: ";
  cin >> tempAge;

  aSheep.setName (tempName);
  aSheep.setAge (tempAge);

  cout << "Enter name of second sheep: ";
  cin >> tempName;
  cout << "Enter " << tempName << "\'s age: ";
  cin >> tempAge;

  anotherSheep.setName (tempName);
  anotherSheep.setAge (tempAge);

  // use operator to find oldest sheep

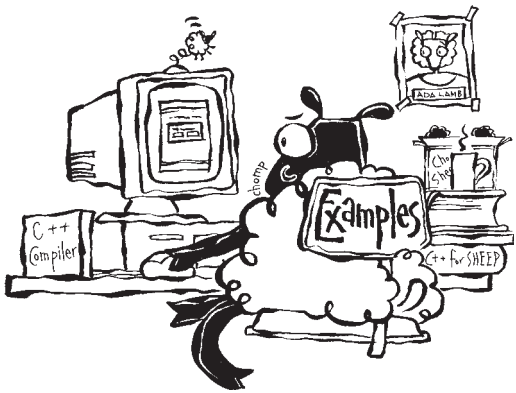
  if (aSheep < anotherSheep) {
    cout << aSheep.getName ()
         << " is oldest" << endl;
  }
  else if (anotherSheep < aSheep) {
    cout << anotherSheep.getName ()
         << " is oldest" << endl;
  }
  else {
    cout << "The sheep are the same age" << endl;
  }
```

```
    return 0;
}
```

It is possible to overload any other operators including the << and >> operators used for input and output. This idea means that classes can be written to work more intuitively and can be made to appear more like the built-in data types.

There's more!

That really is all the C++ in this book. There is still much more to learn, though. Many C++ reference books run to well over a thousand pages and the language is still growing.



There are no examples this time; there were quite enough examples in this chapter anyway! The exercises give you some ideas for some things that you might like to find out now.



20.1 There is a built-in C++ data type called `double`. Find out what it is and how it is different from the data types you have used so far. What other built-in data types are there?

20.2 Find a book on C and investigate the C facilities for formatted output. You can use these in C++ if you include the `cstdio` library.

20.3 Find out how C stores strings. What functions for handling strings does C have? Can you explain what `char **argv` means when command line arguments are being used?

20.4 It is possible to overload the << and >> operators. Find out how to do this and experiment with adding these to a class that you have written.

20.5 Add an "is equal to" (==) operator to the *Sheep* class from this chapter. Write a program to test it.

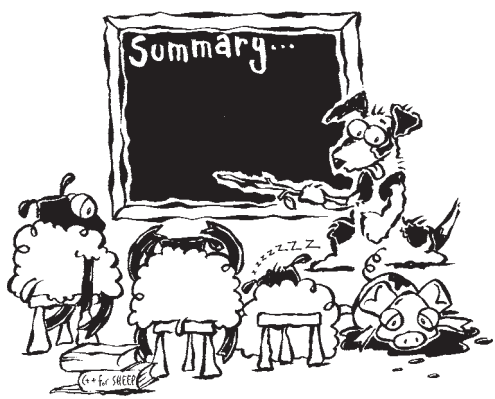
20.6 Write a program that sorts of vector of sheep into order based on their age. Use an overloaded operator.

20.7 Write a program that sorts a file of words into alphabetical order. The input and output should use the same file.

20.8 C++ was originally developed by Bjarne Stroustrup. Visit his web page and read what he has to say about the language.

20.9 What is Java? Find a book on Java and see how much you can understand. How does it differ from C++? What looks much the same?

20.10 Write a program. It can do anything you like as long as it uses most of the C++ that you have seen and used up to now. When you have finished and tested this program, write another. Remember that the only way to become a good programmer is to practise.



This is the end of the book.

Congratulations on getting this far. You'll probably be quite relieved to have made it here. You'll certainly have seen and learned a lot. Remember that learning to program is difficult, and if you've got this far you've achieved something that should make you rather proud.

This chapter has shown you just a few of the other features that are available in C++. You should now be able to find out more about these and also about the ones that haven't been mentioned. You should be able to understand and use a C++ reference book and you should be able to develop more and more as a programmer.

In a way, the end of this book is just a beginning. Programming is a skill that develops with practice. Even the most experienced programmer is always learning. Programmers will always be learning a new language or a new

technique. As computers develop in the future so will the languages that are used to program them. There is no end to this trend in sight. Hopefully you are now in a good position to start to become a programmer.

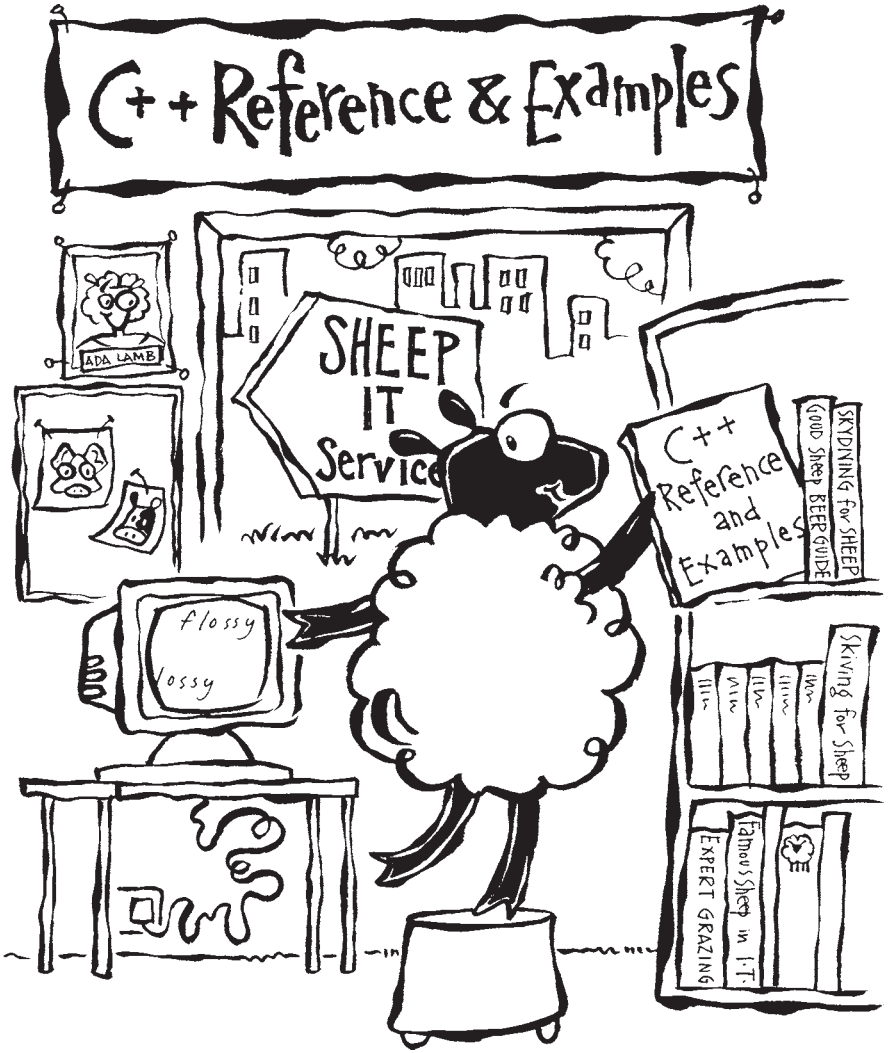
Remember that you have not just learned C++. You have learned programming. Virtually all of the ideas and techniques you have seen and used in C++ are also available in other languages. The syntax will be slightly different, but that is all that you will have to learn when you come to a new language. You'll know what you need to do and will just have to look up the syntax. The hard work is done.

What you need to do now is practise. When you've done that, do some more. The more you write programs the better you become at it, and the easier and more natural it becomes. Many programmers find programming an extremely rewarding and creative process. Hopefully you'll come to believe that too.

My job here is done. Now it's down to you. Whether you're going to go and on learn more C++ or whether it's going to be a different language I wish you the best of luck. Remember to keep on learning.

Onward!




C++ Reference & Examples
ADA LAMB
SHEEP
IT
Service
C++
Reference
and
Examples
SKYDIVING for SHEEP
GOOD Sheep BEER GUIDE
flossy
Skiving for Sheep
Fatmou's Sheep in I.T.
EXPERT GRAZING

Every programmer, even the most experienced, works with a handy language reference beside the computer. No programmer can be expected to remember all the fiddly little details of the syntax of a modern programming language, and it is essential that you get used to using a reference. In the future that might be a book describing all of the C++ language, but for the moment this section will do.

This section gives you examples of all the C++ you've seen in this book, all arranged in the same place. It's in just about the same order as the book and the sections are numbered (there's a small contents list at the start) so that you can (hopefully!) easily find the example you need. If you're lucky, some of the examples here will slot right into the program you're trying to write.

If you can't find what you need here you can also try looking through the solutions to the exercises. One of the best ways of working out how to write a program is to look at a program that does something similar; you might even be able to adapt something. Failing that, there's always the index!

Contents

1. Comments 338
2. Basic structure 339
3. Defining variables 339
4. Defining constants 339
5. Types 339
6. Converting between types 339
7. Assignments 340
8. Operators 340
9. Output 340
10. Input 341
11. Defining a class 341
12. Using a class 342
13. Implementing a class 342
14. Conditional statements – `if` 343
15. Conditional statements – `switch` 344
16. Loops – `for` loops 345
17. Loops – `while` loops 345
18. Loops – `do ... while` loops 346
19. Method prototypes 346
20. Method parameters – value parameters 347
21. Method parameters – reference parameters 347
22. Non-member functions 347
23. Arrays 348
24. Vectors 349

1. Comments

Any line starting // is a comment.

```
// This is a comment
// So is this
```

Any part of a line after a // is also a comment.

```
x = 0; // This explains why that is complicated
```

2. Basic structure

A C++ program starts with a comment describing its purpose and providing other useful information. The basic structure after this follows a set pattern.

```
#include <iostream>

// Other included libraries or files

// prototypes of non-member functions

using namespace std;

int main ()
{
  // Program Statements

  return 0;
}
```

3. Defining variables

A variable has an identifier (name) and a type.

```
int aNumber;
string aName;
bool finished;
```

The variable can also be given an initial value.

```
int aNumber = 0;
string aName = "Flossy";
bool finished = false;
```

4. Defining constants

Constant values are declared in a similar way to variables. Constant values cannot be changed.

```
const int MaxTurns = 10;
const string Book = "How to Program Using C++";
const float PI = 3.1415;
```

5. Types

The following data types have been used in this book:

`int`	an integer – positive or negative (or 0)
`float`	a floating-point number
`char`	a single character
`string`	a sequence of characters
`bool`	a Boolean value – true or false
`vector`	a collection of other values of the same type

6. Converting between types

The names of these types can be used to convert values between types.

```
int aNumber = 11;
float aFloat;

aFloat = float (aNumber);
```

7. Assignments

Variables are assigned values in assignment statements:

```
aNumber = 10;
aString = "hello";
aFloat = 3.1415;
```

The right-hand side can be an expression.

```
aNumber = 10 + 2;
aString = "hello " + "world";
aFloat = 1.5 + 3.2;
```

Identifiers can appear on both sides of the assignment operator.

```
aNumber = anotherNumber + 10;
aNumber = aNumber + 1;
```

Various shorthand forms for common operations exist.

```
aNumber ++;     // aNumber is incremented
aNumber --;     // aNumber is decremented
aNumber += 3;   // 3 is added to aNumber
aNumber -= 4;   // 4 is subtracted from aNumber
aNumber /= 2;   // aNumber is divided by 2
aNumber *= 4;   // aNumber is multiplied by 4
```

8. Operators

There are four basic arithmetic operators. In order of precedence they are:

/	Division
*	Multiplication
+	Addition (of numbers) or concatentation (of `strings`)
-	Subtraction

These work as expected for numeric data.

```
aNumber = 2 + 2;    // assigns 4
aFloat = 1.5 + 3.7; // assigns 5.2
```

The addition operator concatenates two strings.

```
aString = "Flossy " + "the Sheep";
    // assigns "Flossy the Sheep" (without the quotes)
```

Operators are applied in order of precedence. The order is as in the table above.

```
aNumber = 10 + 6 / 2;   // aNumber is assigned 13
aNumber = 10 / 2 + 3;   // aNumber is assigned 8
aNumber = 10 + 3 * 2;   // aNumber is assigned 16
```

The order can be affected by adding brackets.

```
aNumber = (10 + 6) / 2; // aNumber is assigned 8
aNumber = 10 / (2 + 3); // aNumber is assigned 2
aNumber = (10 + 3) * 2; // aNumber is assigned 26
```

9. Output

Values are displayed on the screen using `cout <<`.

```
cout << "hello";
cout << aNumber;
```

```
cout << "The number is " << aNumber;
```

If quotes are required in the output they are preceded with \.

```
cout << "\"hello\"";
```

A "return" is inserted by `endl`.

```
cout << "hello" << endl;
cout << "hello" << endl << aNumber << endl;
cout << endl;
```

Error messages can be written to the error stream, `cerr`.

```
cerr << "Help! Something awful has happened" << endl;
```

10. Input

Values are read from the user using `cin >>`.

```
cin >> aNumber;
cin >> aString;
```

Input is terminated by white space – a space, TAB or newline.

11. Defining a class

A class is defined in a header file. The name of the class is defined, together with its private and public sections.

```
using namespace std;

class Sheep {
  private:
    // Private Attributes
  public:
    // Public Methods
};
```

Attributes are defined in the private section.

```
#include <string>

using namespace std;

class Sheep {
  private:
    string name;
    int age;
    float value;

  public:
    // Public Methods
};
```

Methods are defined in the public section. There must be a constructor and there are usually selectors and mutators.

```
#include <string>

using namespace std;

class Sheep {
  private:
```

```
    string name;
    int age;
    float value;

  public:
    Sheep (); // Constructor

    // Selectors

    string getName () const;
    int getAge () const;
    float getValue () const;

    // Mutators

    void setName (string);
    void setAge (int);
    void setValue (float);
};
```

Methods that are guaranteed not to change the values of any of the attributes have a `const` qualifier added to the prototype.

The prototype of each method specifies the return type, the name of the method, and the types of any parameters.

12. Using a class

An instance of a class is declared in the same way as any other variable.

```
Sheep flossy;
```

Methods are called by joining the identifier of the instance to the name of the method with a full stop.

```
flossy.setName ("Flossy");
flossy.setAge (6);
cout << flossy.getName () << " is "
     << flossy.getAge () << endl;
```

13. Implementing a class

A class is implemented in an implementation file. The `::` notation is used to associated a method with the class.

The implementation file must "include" the definition in the header file.

```
#include "sheep.h"

// Constructor

Sheep::Sheep ()
{
  return;
}

// Selectors

string Sheep::getName () const
{
  return name;
}
```

```
int Sheep::getAge () const
{
  return age;
}

float Sheep::getValue () const
{
  return value;
}

// Mutators

void Sheep::setName (string newName)
{
  name = newName;
  return;
}

void Sheep::setAge (int newAge)
{
  age = newAge;
}

void Sheep::setValue (float newValue)
{
  value = newValue;
  return;
}
```

14. Conditional statements – `if`

The basic `if` statement tests a condition and executes a statement only if the condition is true.

```
string aName;
int age;
bool finished;

if (aName == "Flossy") {
  cout << "The name is Flossy" << endl;
}

if (age > 10) {
  cout << "The age is greater than 10" << endl;
}

if (finished) {
  cout << "All finished!" << endl;
}
```

The optional `else` clause defines the statements to be executed if the condition is false.

```
if (aName == "Flossy") {
  cout << "The name is Flossy" << endl;
}
else {
  cout << "The name is not Flossy" << endl;
}
```

```
if (age > 10) {
  cout << "The age is greater than 10" << endl;
}
else {
  cout << "The age is 10 or less" << endl;
}

if (finished) {
  cout << "All finished!" << endl;
}
else {
  cout << "Onward!" << endl;
}
```

More than one condition can be tested.

```
if (aName == "Flossy") {
  cout << "The name is Flossy" << endl;
}
else if (aName == "Freda") {
  cout << "The name is Freda" << endl;
}
else {
  cout << "The name is not Flossy or Freda" << endl;
}
if (age > 10) {
  cout << "The age is greater than 10" << endl;
}
else if (age < 10) {
  cout << "The age is less than 10" << endl;
}
else {
  cout << "The age is 10" << endl;
}
```

Conditions can be combined using the Boolean operators: *and* (&&), *or* (||), and *not* (!).

15. Conditional statements – `switch`

The `switch` statement is a shorthand form of the `if` statement where the possible values for a variable can be listed and different statements are executed for each value.

The variable must be of an ordinal type – integer or character.

```
char direction;

switch (direction) {
  case 'n': cout << "North" << endl;
            break;
  case 's': cout << "South" << endl;
            break;
  case 'e': cout << "East" << endl;
            break;
  case 'w': cout << "West" << endl;
            break;
}
```

The `default` case handles unexpected values.

```
switch (direction) {
  case 'n': cout << "North" << endl;
            break;
  case 's': cout << "South" << endl;
            break;
  case 'e': cout << "East" << endl;
            break;
  case 'w': cout << "West" << endl;
            break;
  default:  cout << "Invalid direction! << endl;
}
```

More than one value can be associated with a case.

```
switch (direction) {
  case 'n':
  case 'N': cout << "North" << endl;
            break;
  case 's':
  case 'S': cout << "South" << endl;
            break;
  case 'e':
  case 'E': cout << "East" << endl;
            break;
  case 'w':
  case 'W': cout << "West" << endl;
            break;
  default:  cout << "Invalid direction! << endl;
}
```

16. Loops – `for` loops

A `for` loop is determinate. It executes a number of times that can be determined before the first execution.

The loop is controlled by an initial statement, a continuation condition, and a statement that is executed each time the loop runs.

```
// Loop 10 times
for (int counter = 0; counter < 10; counter ++) {
  cout << counter << endl;
}

// Loop 10 times, but backwards
for (int counter = 10; counter > 0; counter --) {
  cout << counter << endl;
}
```

17. Loops – `while` loops

A `while` loop continues to execute while some condition is true. It is indeterminate.

```
while (!finished) {
  // run the program
}

while (!finished && attempts < 10) {
  // run the program
}
```

The value of the condition must change inside the loop, or the loop will never terminate.

```
while (!finished) {
  // run the program

  if (answer == 'y') {
    finished = true;
  }
}

while (!finished && attempts < 10) {
  // run the program

  attempts ++;
}
```

If the condition is initially false the statements inside the loop are never executed.

```
bool finished = true;

while (!finished) {
  // never executed
}
```

18. Loops – `do ... while` loops

A `do ... while` loop executes while a condition is true. It is indeterminate.

```
do {
  // run the program
} while (!finished);
```

The value of the condition must change inside the loop:

```
do {
  // run the program

  finished = answer == 'y';

} while (!finished);
```

The condition is tested after the statements inside the loop have been executed. This means that the loop is always executed at least once.

```
bool finished = true;

do {
  // executed once
} while (!finished);
```

19. Method prototypes

The prototype of a method specifies the name of the method, the type of value returned by it, and the types of its parameters. The types of the parameters are listed in brackets, which are left empty if there are none.

```
int getAge ();
void setAge (int);
float getValuePlusTax (float);
```

A method that does not change any of the values of any of the private attributes is defined to be constant.

```
int getAge () const;
```

A method that does not return any value is defined to return `void`.

20. Method parameters – value parameters

Methods process values. These can be literal values or variables.

```
Sheep flossy;
int age = 10;

flossy.setAge (10);
flossy.setAge (age);
```

Methods may alter the values passed to them, but the changes are not made to the variable in the calling program.

If a method has more than one parameter the values must be supplied in the correct order.

```
void printInColumns (float, int);

flossy.printInColumns ("baa", 4);   // wrong!
flossy.printInColumns (2.5, 4);     // correct
```

21. Method parameters – reference parameters

Methods may also change values that are then available to the main program. Reference parameters are denoted in the prototype with an & symbol.

```
void runningTotal (float&);
```

Reference parameters are passed to a method in the same way as value parameters.

```
float total = 0;
flossy.runningTotal (total);
```

A reference to the variable is passed, so the same memory location is used in the method as in the calling program. A change to the value in the method thus alters the value in the calling program.

22. Non-member functions

Non-member functions are not associated with any particular class. They are specified by a prototype as for a method.

```
void swap (int&, int&);
string longest (string, string);
void printSquare (int, char);
```

A program that uses a non-member function must include the prototype. They are defined in a similar way to methods.

```
void swap (int& x, int& y)
{
  int temp = x;

  x = y;
  y = temp;

  return;
}
```

```
string longest (string s1, string s2)
{
  if (s1.length () > s2.length () ) {
    return s1;
  }
  else if (s1.length () < s2.length () ) {
    return s2;
  }
  else {
    return s1;
  }
}

void printSquare (int side, char c)
{
  for (int down = 0; down < side; down ++) {
    for (int across = 0; across < side; across ++) {
      cout << c;
    }
    cout << endl;
  }

  return;
}
```

23. Arrays

An array stores a collection of values of the same type. The type and the number of values is specified in the declaration.

```
const int NumOfScores = 10;
int bestScores[100];
int scores[NumOfScores];
```

Each value in the array is called an element and can be referenced by its index. The first element is at index 0.

```
cout << "The first element is " << scores[0]
     << endl
     << "The last element is " << scores[9];
```

An array can be passed to a function or method. It is always passed as a reference parameter. The number of values in the array must also be passed.

```
int highestElement (int[], int);
void sort (string[], int);
```

A common operation is to scan an array using a `for` loop to examine each element in turn.

```
int highestElement (int values[], int number)
{
  int highest = values[0];
  for (count = 1; count < number; count ++) {
    if (values[count] > highest) {
      highest = values[count];
    }
  }
}
```

If a method or function should not be allowed to alter the values in an array, the array can be passed as a constant parameter.

```
int highestElement (int[] const, int);
```

24. Vectors

A vector is similar to an array. The main difference is that its size is not fixed. The declaration specifies the type that is stored in the vector and, optionally, an initial size.

```
vector <int> marks;
vector <string> names;
vector <float> averages (10); // initial size
```

A vector is a class. A value is added to a vector using the `push_back` method.

```
marks.push_back (10);
names.push_back ("Flossy");
averages.push_back (12.56);
```

An element in a vector can be accessed using the same notation as for an array.

```
cout << "The first mark is " << marks[0] << endl;
cout << "The first name is " << names[0] << endl;
```

The `size` method returns the number of elements in the vector.

```
cout << "There are " << marks.size () << " marks "
     << endl
     << "The last mark is " << marks[marks.size () - 1]
     << endl;
```




Appendix A - Further Reading, Web-Sites and Other Resources
APA LAMB
SHEEP IT services
C++ Ref & Examples
search

Since we're now at the end of the book, the question rather arises of where to go next[1]. This section contains some ideas. Here you'll find pointers to books and web sites that should help you along the way as you learn more about programming and more about computing in general.

All the web sites in this section worked at the end of January 2003. The web is a fast-changing place, and it is quite possible that by the time you read this some of them will have disappeared or moved on. A good search engine should help you find out where it's gone. If you find a web site that's not working, you can email me at *tony@comp.leeds.ac.uk*, especially if you've also found a decent replacement! All the web sites are also linked from this book's web site, together with any changes.

Most people's search engine of choice these days seems to be Google at *http://www.google.com/*, not least because it's refreshingly advertisement-free. That's the one I used to find the sites in this section.

Other useful search engines are:

AltaVista	*http://www.altavista.com/*
Excite	*http://www.excite.com/*
Lycos	*http://www.lycos.com/*
Yahoo	*http://www.yahoo.com/*

More about C++ (and C)

The best place to start if you want to learn more about the C++ language is probably Bjarne Stroustrup's home page:

http://www.research.att.com/~bs/

As well as gazing on a picture of the man himself (and checking the handy guide to pronouncing his name), here you can find Stroustrup's own writings on the language. Find out how and why he chose to add object-oriented features to C++ and just what he thinks about how the language has developed since then. There's also a collection of C++ links that will lead you to just about anything else that you want to know.

Stroustrup's own book about C++ is worth getting hold of if you want the full details from the man who should know most:

Bjarne Stroustrup (1997). *The C++ Programming Language*. Addison-Wesley, Reading, MA.

Opinion seems to be divided on the merits of this book. Some programmers swear by it, while others would never use it.

There are many other C++ books. The best way to choose the one that's best for you is to spend some time in a shop leafing through them. You can also find a collection of reviews provided by the Association of C and C++ Users at:

http://www.accu.org/

You will have gathered by now that I'm quite keen on things that are free. Happily, there are free books on C++. The most widely used of these is probably Bruce Eckel's *Thinking in C++*; the second edition of this is available in an electronic form at:

---

1 You may well fancy the pub, a nice café, Hawaii or Basingstoke, but I'm talking C++ here.

`http://www.mindview.net/Books/TICPP/ThinkingInCPP2e.html`

Another free programming book is Allen Downey's *How to Think Like a Computer Scientist*. This is available for a few languages, but the C++ version seems a little sketchy. You can find it at:

`http://www.ibiblio.org/obp/thinkCS.php`

C++ was, of course, an extension of C. The seminal book on C is undoubtedly:

Brian W. Kernighan and Dennis M. Ritchie (1988). *The C Programming Language*. Prentice Hall, Englewood Cliffs, NJ.

If you get hold of a copy of this you'll find out that C does indeed look rather like C++; it's just that there aren't any classes around. Remember that all C programs are also valid C++ programs and you can make use of some of the neat features of C in C++ (in particular, many programmers prefer to use C's approach to formatted output). It's certainly worth spending a bit of time getting used to C.

The home pages of Kernighan and Ritchie provide a good bit of history of the development of C. The contribution of these two (K and R, as they are generally known) is hard to overstate. Both K and R work at Bell Labs in New Jersey and their pages are:

`http://www.cs.bell-labs.com/who/dmr/`
`http://www.cs.bell-labs.com/who/bwk/`

Ritchie's collection of papers about the history of C (and BCPL) is certainly worth a read.

These three home pages will also reveal the unsettling truth that all the developers who brought us C and C++ have rather splendid beards. There is no truth in the rumour, though, that a beard is essential to program effectively in C++.

Famous people

The names of a few other famous people in the history of computing have been used in some of the examples. These are some people that you really should know something about:

Charles Babbage The "Grandfather of Computing". Built (or more accurately attempted to build) the first mechanical devices that shared many of the characteristics of modern computers.

`http://ei.cs.vt.edu/~history/Babbage.html`

Grace Hopper A pioneer of programming and compilers. Curiously given the first computing Man of the Year award in 1969.

`http://www.sdsc.edu/ScienceWomen/hopper.html`

Ada Lovelace Generally thought of as the first computer programmer, although she lived long before anything that we might recognise now as a computer was built.

`http://www.sdsc.edu/ScienceWomen/lovelace.html`

John von Neumann	A pioneer in computer architecture – the von Neumann Architecture. `http://ei.cs.vt.edu/~history/VonNeumann.html`
Linus Torvalds	The man behind the free Linux operating system. A famous Finn. `http://www.tuxedo.org/~esr/faqs/linus/`
Alan Turing	A founder of artificial intelligence, probably now most famous for his work at Bletchley Park in the Second World War. `http://www.turing.org/`
Maurice Wilkes	Developed many fundamental aspects of computer hardware and programming. `http://ei.cs.vt.edu/~history/Wilkes.html`

As usual, a decent search engine will turn up many more pages of information about these people.

There are also many sites offering information about more famous people and more about the history of computing in general. The site at Virginia Tech is a good starting point if you want to find out more:

`http://ei.cs.vt.edu/~history/`

Reader's choice

That's it for further reading.

If you want more, you'll have to send it to me. On the book's web site you'll find a form to submit your favourite, useful, related-to-programming-in-some-vague-sort-of-way web site. Send it to me and I'll add it to the links – your chance to get a little bit of fame!

Software

This section describes how to get hold of all the software you need to get started writing C++. The best part is that all the software described here is completely free!

All the software in this section can be downloaded free of charge from the web. If this causes a problem, you can get it all on a set of CDs as part of the excellent Brighton University Resource Kit for Students (BURKS). Details of BURKS are at:

`http://burks.bton.ac.uk/`

BURKS includes a very large collection of software and other useful stuff, including a full distribution of Linux, a free version of the Unix operating system. The web site also contains a mirror of the contents of the CD, so you can download everything from there too. If you choose to buy the CD, please be sure to tell John that I sent you.

Most of the Linux versions of this software will also be provided as a standard part of one of the many distributions of Linux. Links to the web sites of the providers of all of these distributions are maintained at:

`http://www.linux.org/`

A good place to find out more about free software is the home page of the Free Software Foundation:

`http://www.fsf.org/`

Of course, a computer needs an operating system. Most free software is, very reasonably, written for free operating systems. In practice this means that most free software is written for Linux, and other free versions of the Unix operating system. Do not panic if you're a user of a less free operating system such as Microsoft Windows; there are usually versions available for these too (although sometimes they're not the latest version).

You can find links to all this free software on the web site. You'll also find a form to tell me about any other useful software for C++ programming that you might find so that I can add it to the list.

Compilers

The first thing that you need is obviously a compiler. The most popular free C++ compiler is *g++*, part of the GNU Compiler Collection:

`http://gcc.gnu.org/`

The latest version of the compiler for Linux (and other flavours of Unix) can be downloaded from here. You'll have seen from the comments that this is the compiler I used for all the programs in the book.

The most popular version of *g++* for Windows operating systems is probably *DJGPP*, provided by Delorie Software at:

`http://www.delorie.com/`

The BURKS collection has some others.

Editors

Now for something to edit the source code with. In the Unix world the most popular editor is *vi*. A fine version of this is *Vim*, available for a vast range of operating systems (including Microsoft Windows) at:

`http://www.vim.org/`

New programmers often find the windows-based version of *Vim* (called *gvim*) especially useful. *Vim* has many features that make programming easier; it will highlight different statements in helpful colours, and it will take care of indentation automatically. All the programs in this book were written using either *Vim* or *gvim*.

Vim is not strictly speaking free software. It is *charityware*. If you download *Vim* and find it useful you are invited to make a donation to help needy children in Uganda.

A popular editor for Windows platforms is the Programmer's File Editor (PFE). Development of PFE has now been discontinued, but the last version can still be found:

`http://www.lancs.ac.uk/people/cpaap/pfe/`

If you fancy something a little more sophisticated for your programming you could try out an IDE. Dev-C++ written by the splendidly named Bloodshed Software is one example:

`http://www.bloodshed.nu/`

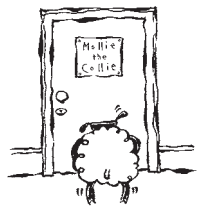
Delorie Software provide another, called DFE95. Both these, and more, are available on BURKS.

This book

The programs for this book where developed using *g++*. The source code was created using *Vim*. All this happened using various versions of the Red Hat distribution of the Linux operating system.

The words you are reading now were typed using Microsoft Word 2000, running on Microsoft Windows 2000.

Getting the programs into and out of Word was a nightmare. It is not something that I would recommend.



Other programming languages

It's quite likely that you will soon want to move on to programming in another language. The principles that you have learned with C++ should see you well placed to go on to program in other languages – the obvious candidates are two other object-oriented programming languages: Java and C#.

This section also points you in the right direction to find out more about programming languages in general. In particular, Python is an interesting new language that is well worth a look.

Java

You've probably heard of Java, and it may well be the next language that you move on to use. Java is, like C++, an object-oriented programming language. The main difference is that Java is seen by many to be more "pure" (essentially everything in Java is an object), while C++ is more like a non-object-oriented language (C) with object-oriented features added.

With experience in C++, you should be able to recognise what simple Java programs are doing. Much of the syntax is very similar; the designers of Java had obviously seen C++. Here's the traditional first program in Java:

```
/*
** Hello.java
**
** The Traditional First Program.
**
** Author:    GPH
** Date:      15th September 2002
** Platform:  java, Linux
*/

public class Hello
{
  public static void main(String args[])
  {
    System.out.println("hello, world");
  }
}
```

You can see that Java is also based around the idea of a class and that some of the keywords used in Java are the same as those in C++. Although this looks rather like a class definition, it is in fact a complete Java program; everything in Java is a class, even the program.

The traditional second program written in Java starts to show where the two languages are different, but there should still be plenty here that you can recognise or at least guess at the purpose of.

```
/*
** HelloName.java
**
** The Traditional Second Program.
**
** Author:    GPH
** Date:      15th September 2002
```

```
** Platform:  java, Linux
*/

import java.io.*;

public class HelloName
{
  public static void main(String args[])
  {
    String name = "";
    BufferedReader cin;

    System.out.print("Who are you? : ");
    cin = new BufferedReader
         (new InputStreamReader(System.in));

    try
    {
      name = cin.readLine();
    }
    catch(IOException ioe)
    {
      System.err.println("Error reading input.");
      System.exit(1);
    }
    System.out.println("hello, " + name);
  }
}
```

Java was developed by a team at Sun Microsystems in a project that started in the early 1990s; the original aim was to develop tools for programming intelligent consumer electronics. You can read all about its early history on Sun's web site:

http://java.sun.com/features/1998/05/birthday.html

This history is part of Sun's main Java site:

http://java.sun.com/

which contains just about all the information about Java you could possibly want.

A key feature of Java is its portability. It is claimed that a correct Java program will work unaltered on any computer (or other device) capable of running Java. It should be obvious why portability of this type would be of great interest to all sorts of software companies. Java is also seen by many as the language of the Internet; you have almost certainly seen web sites making use of Java applets for interactive content. Java is, however, not to be confused with JavaScript.

James Gosling, one of the leaders of the project that developed Java, also has a page that provides his version of the history:

http://java.sun.com/people/jag/green/

Gosling's home page is also worth visiting:

http://java.sun.com/people/jag/

not least for the picture of Bill Gates and the pie.

Reassuringly, Gosling also has a splendid beard.

C#

You might also have heard of C# (pronounced C-sharp). C# is closely related to C++ and Java, and is worthy of quick mention here. C# is a proprietary product of Microsoft; it is the programming language associated with Microsoft's .NET initiative. Like Microsoft's other visual programming languages, C# is written using a complete IDE (you have probably seen or used tools such as Access or Visual BASIC).

C# is quite a controversial topic. Some have argued that it is simply Microsoft's attempt to compete with Java; Microsoft obviously has a proprietary operating system and could be expected to be worried by a language that would run on many different systems. Some have gone further and suggested that C# is little more than a copy of Java made specific for the Microsoft platform. Others, on the other hand, state that the emergence of C# and Java at roughly the same time is little more than coincidence. There are plenty of people who will argue for each of these.

You can make your own mind up. Microsoft's own site contains all you could need to know about C#, as well as handy pointers on where to buy it:

http://msdn.microsoft.com/vcsharp/

A search on any web search engine will lead you to many of the arguments!

Of some significance may well be the image on:

http://www.microsoft.com/billgates/

No beard, you see?

Python

A final language worth a mention in its own right is Python. Python is, like Java, a platform-independent language and implementations of Python are free. You can find out all about it and download a copy at the Python web site:

http://www.python.org/

In particular, the Beginners' Guide at:

http://www.python.org/doc/Newbies.html

will help you on your way to learning some Python. There is also an excellent free book by Jeff Elkner and others describing all the Python you might need available:

http://www.ibiblio.org/obp/thinkCSpy/

and an equally excellent free tutorial by Alan Gauld here:

http://www.freenetpages.co.uk/hp/alan.gauld/

Python differs from C++ in that it is a scripting language. This means that Python programs are not compiled, but are interpreted as they are executed. Python programs can also be written interactively.

Python is quite a new language, but it is gaining in use. A particular attraction is that, since it is interpreted, it is especially suitable for developing quick prototypes of programs before they are rewritten in another language. Many recent applications have used Python, including the Google search engine and several feature films.

The history of Python is described on the home page of the author, Guido van Rossum:

http://www.python.org/~guido/

He has no beard.

Other languages

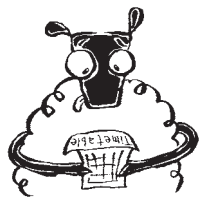
Computer programming languages are rather fascinating things. They have developed a complex family tree, even though they've only been around for a relatively short time. If you want to find out more about them, a good start is Éric Lévénez's excellent programming language page:

http://www.levenez.com/lang/

Equally fascinating is the splendid "Bottles of Beer" page at:

http://99-bottles-of-beer.ls-la.net/

where you can find programs to generate the lyrics to a well-known song in (as at January 2003) 487 different programming languages.



Glossary

The following terms are used in the book; their definitions are reproduced here.

Alternative, perhaps better, definitions can be found in one of the many dictionaries of computing terminology on the web. One of the best is FOLDOC, the Free On-Line Dictionary of Computing, available in many places including:

`http://wombat.doc.ic.ac.uk/foldoc/`

There are many more.

For a different slant, the infamous Jargon File provides many slightly alternative definitions:

`http://www.tuxedo.org/~esr/jargon/`

accessor See *selector*.

algorithm The "recipe" that describes how a *program* should work. Often written in *pseudo-code*.

argument See *parameter*.

array A structure for storing a collection of related values. Inherited from C. See also *vector*.

assignment operator `=`. Used in an *assignment statement*. Not to be confused with the comparison operator `==`.

assignment statement A statement that assigns a value to a *variable*.

attribute A single value for a feature of an *object type* that is of interest in a problem area.

backslash The \ character, normally found on the keyboard to the left of the Z key.

base class A *class* from which others inherit *attributes* and *methods*.

beta testing The testing of an early release of a *program* by a selected *user*. The purpose is to find *bugs* that can only be discovered by actual use.

black box testing The process of *testing* a *program* without examining the *source code*.

body The part of the *program* inside the main function.

Boolean A value that is either true or false.

Boolean expression An *expression* that combines one or more *Boolean values* (or one or more *expressions* that yield *Boolean values*) to give a resulting *Boolean value*.

Boolean operator An *operator* used to build a *Boolean expression* – and, or, not.

Boolean value See *Boolean*.

boundary case A *test case* that concentrates on a particular set of *test data* that might represent an error as values change from acceptable to unacceptable.

braces Brackets used to enclose connected *statements* in a *program* – { and }.

Bubblesort A common and simple *sorting algorithm*.

bug An undocumented feature in a *program* that means that the *program* does not work as expected. Often discovered by a *user*.

built-in function A *function* provided as part of the C++ language. Many C++ functions are actually inherited from the C language.

call A *function call* or a *method call*, or to invoke either.

calling function A *function* that calls a *method* or another *function*.

calling program A *program* that calls a *method* or *function*.

cast A *function* that converts a value from one *data type* to another, most usually from one numeric type to another.

class In C++, an object type.

code See *source code*.
code walkthrough A form of *white box testing* where one person, probably the *programmer*, goes systematically through a *program* explaining each *statement* to another person.
comment Lines in a *program* that have no effect other than to convey information to a human reading the *program*. Preceded by // in C++.
compilation The process of generating an *executable* version of a *program* from some *source code*.
compiler The software used for *compilation*.
concatenation The adding of two strings, the second after the first, to form a third string. Similar to addition of numbers.
condition An assertion that is either true or false. Used in a *conditional statement* to control the flow of a *program*.
conditional statement A *control statement* that evaluates a *condition* and, based on the truth or otherwise of this condition, chooses between two or more possible execution paths for a *program*.
constant A value in a program, not unlike a *variable*, that does not change. Also has an *identifier* and a *type*.
constant parameter A *parameter* passed as a *reference parameter* but which cannot be changed by the *method* or *function*. Used when passing *arrays*. Sometimes called a constant reference parameter.
constructor A special *method* that creates an instance of a *class*. Must be declared in the *public part* of the definition of a *class*.
control statement A *statement* that controls the flow of other *statements*. See *conditional statement* and *loop*.
control variable A *variable* used in a *conditional statement* (normally a `switch` statement) or in a *loop* to control the *loop*'s execution.
data hiding The practice of implementing a *class* so that all *attributes* are defined in the *private part*. This effectively means that the details of the storage of the *attributes* can be ignored by a *programmer* using the *class*.
data type The kind of value that can be stored in a particular attribute or variable – *integers*, characters, *strings*, *vectors*, for example.
debugger Software that allows a *programmer* to examine a *program* as it is running in order to find *bugs*.
debugging The process of finding and correcting *bugs* in a *program*. Often carried out with the aid of a *debugger*.
decrement To subtract one from.
derived class A *class* inheriting attributes and methods from a *base class*.
determinate loop A `for` *loop*. The number of times that the *loop* will be executed can be accurately forecast before the *loop* is executed the first time.
directory The name for a *folder* on a Unix system. See *folder*.
driver A *program* written to allow a *class* to be tested.
driver program See *driver*.
dry run To simulate the execution of a *program* using pen and paper as part of the process of *testing*.
editor The software used to create *source code* in a file.
element A value in an *array* or *vector*. Can be accessed via its *index*.
executable A version of a *program* that can be executed by the computer. The result of the process of *compilation*.
expression A combination of *literal values* or *variables*.
file extension The part of a file's name, usually after a full stop, that indicates what the type of the file is.
floating-point number A number that includes a decimal part.
flow of control The order in which the *statements* in a *program* are executed. Flow is controlled by *conditional statements* and *loops*.
folder The name for a *directory* on a Microsoft Windows system. See *directory*.

function A self-contained unit of a *program* that achieves a particular effect and returns a value.
function call A *statement* that causes a *function* to be executed. See also *calling program*.
hard copy See *listing*.
header block A set of *comments* at the start of a *program* that provides information about the purpose of the program, the name of the file containing the *souce code*, the name of the author, the date the program was written, the intended computer system, and possibly more.
header file The file, usually with a `.h` *file extension*, that contains the definition of a C++ class. The file specifies the *types* of the *attributes* of the class and provides *prototypes* for the *methods*.
identifier The name of a variable.
implementation file The file, usually with a .cc *file extension* that contains the C++ code to implement the *methods* of a *class*.
increment To add one to.
indentation The practice of leaving blank spaces at the start of lines of a *program* and so lining up connected *statements*. Consistent indentation is a key element of good programming *style*.
indeterminate loop A `while` or `do ... while` *loop*. A *loop* that is controlled by a *condition*, causing *statements* to be executed while some *condition* is true or until some *condition* becomes false.
index A position in an *array*, *vector*, or *string*.
infinite loop A *loop* that never terminates – almost always an error.
integer A whole number.
interactive development environment Or IDE. A piece of software that combines an *editor* and *compiler* (and possibly other tools such as a *debugger*) into a convenient system for *program* development.
interface The specification of the *methods* of a *class* (usually found in the *header file*) that specifies how the *class* can be used in a *program*.
linker The software that links the *object code* from a *program* with the *system libraries* to provide a final version of a *program* that can be executed by a computer.
listing A print out of the *source code* of a *program*; an invaluable aid in *debugging*.
literal value A simple value used in an assignment: 1, 1.5, "alan turing".
local variable A *variable* that is declared within a *function* or *method* and which exists only within that *function* or *method*.
loop A *control statement* that causes a *statement* (or a set of *statements*) to be executed a number of times. See *determinate loop* and *indeterminate loop*.
method A procedure associated with an object type that processes the *attributes* of an instance of the object type.
method call A *statement* that causes a *method* to be executed. See also *calling program*.
methodology A structured way of carrying out some task (such as analysis of a problem).
mutator A *method* that changes the value of an *attribute* of an *object*.
non-member function See *function*.
object Something of interest in a problem area that a *program* must be able to process.
object code An intermediate form of a *program*, produced as part of the *compilation* process. Object code is linked with *system libraries* to provide a final *executable* version of a *program*.
object instance See *object*.
object-oriented language A programming language that supports the declaration and implementation of *objects* and *programs* to manipulate *objects*.
object-oriented programming Programming in an *object-oriented language*. Developing the *program* involves first specifying and implementing the objects in the problem area.
object type A group of *objects* in a problem area that share the same characteristics.

operating system The *program* (or collection of *programs*) that carries out the basic tasks of the computer.

operator A symbol that can be used to build up an *expression* by combining *variable identifiers* and *literal values*. /, *, + and -. See also *Boolean operator*.

operator precedence The order in which *operators* in an *expression* are applied. Basically, division, multiplication, addition, and finally subtraction.

parameter A value processed by a *method* or *function*.

portable A property of a *program* that can be compiled and run, unchanged, on a variety of *operating systems*.

preamble The part of a program before the start of the main function.

precedence See *operator precedence*.

private part The part of the definition of a *class* that describes the inner workings of the *class*. These are not available for use in *programs*. Usually, this is the *attributes* of the *class*.

program A set of C++ *statements* that together completes some task.

programmer A skilled craftsperson who develops a *program*.

prompt A string displayed to indicate to a *user* that a *program* requires a value to be input, and to indicate what this value is to represent.

prototype A specification of a *method* or *function* that includes the name of the *method* or *function*, the *type* of value returned, and the *types* of any *parameters*.

pseudo-code A structured way of writing out an *algorithm*. Pseudo-code lies somewhere between natural language and a programming language; it is written in "English" but in a form that is precise and unambiguous.

public interface See *interface*. The *interface* of a *class* defined in the *public part* of the definition.

public part The part of the definition of a *class* that describes the *interface* of the *class* that is available to be used in *programs*. This is usually the *methods* of the *class*.

recursion See *recursion*.

reference parameter A *parameter* passed to a *method* or *function* so that the final value of the *parameter* is available in the *calling program* or *calling function*.

return type The *data type* of a *return value*.

return value The value produced (and returned) by a *method* or *function*.

selector A *method* that returns a value for a particular *attribute* of an *object*. Sometimes called an *accessor*.

software engineer One who engages in *software engineering*. Sometimes used incorrectly as a synonym for *programmer*.

software engineering The whole process of producing a working computer system, including *programming*, *testing*, documenting and more.

sorting algorithm An *algorithm* for sorting a collection of values, usually stored in an *array* or *vector*. See for example *Bubblesort*.

source code A *program* written, usually in a text file, by a *programmer*. This is the form of the *program* that is understood by a human.

statement A line of C++ that achieves some effect.

stream The mechanism that C++ *programs* use to handle input and output.

string A sequence of characters, usually spelling out some word or other sequence with some meaning.

style The way in which a *programmer* lays out the *source code* of a *program*. Consistent style is very important.

system libraries Code provided as part of a programming language to carry out various common tasks. These libraries will be different on different *operating systems*, thus allowing a *program* to be *portable* between *operating systems*.

test case A set of input values and corresponding output values that can be used as part of a *test plan* to test a program.

test data The values supplied as input in a *test case*.

tester One who applies a *test plan*. Sometimes, but not always, also the *programmer*. Requires cunning.

test harness A form of *driver program* used to test a *class*.
testing The process of applying a *test plan* to a *program* in order to determine whether or not the program appears to work correctly.
test plan A collection of *test cases* that are together sufficient to demonstrate that a program works correctly.
text editor See *editor*.
transaction Something that happens in a real world problem that changes or transmits some values. In analysis, often a sign of a *method* or *function*.
truth table A table used to show the results of a *Boolean operator* or *Boolean expression*.
type See *data type*.
typical values A *test case* that represents a (perhaps randomly chosen) set of *test data* that a *program* might usually process.
user A person who takes a *program* written by a skilled *programmer* and manages to use it in all sorts of incorrect and unexpected ways. See also *bug*.
value parameter A *parameter* passed to a *method* or *function* as a *literal value*.
variable A value in a C++ *program* that varies as the *program* executes.
variable declaration A *statement* that creates a *variable*, specifying its *identifier*, its *type* and (optionally) its initial value.
variable parameter See *reference parameter*.
vector A C++ structure for storing collections of values. See also *array*.
white box testing The process of *testing* by examining the *source code* of the program.
white space A space, tab, or return character.




Index
Freda
Fiona
Fran

Welcome to the index. The following conventions are used:

- Where a topic is mentioned on many pages but is covered in detail in one place, a page number in **bold** indicates the main coverage. Sometimes the main coverage is in two places.
- A page number in *italics* indicates a complete program or class definition or implementation containing an example of a programming construction or an illustration of a subject. For obvious reasons this list is not complete, but it is representative; each reference is a good example of the topic.

Many of the terms in this index are also defined in the Glossary, starting on page 360. The C++ Reference, starting on page 338, also contains examples of all the main programming ideas.

All complete programs, header files and implementation files are also listed by the file name given on the first line of the comment.

+ (addition) 81, **83**
+ (concatenation of strings) 81
- (subtraction) 83
* (multiplication) 83
/ (division) **83**, 84–5
++ (increment) 82, **83–4**
-- (decrement) 84
+= (add to) *69*, 84, 88, *101–2*
-= (subtract from) 84, 88, *101–2*
*= (multiply by) 84
/= (divide by) 84
% (modulus) 89
= (assignment) *76*, 78, **81–2**, 137, 192
== (comparison) **166–7**, 168, 170, 192
!= (not equal to) **166–7**, 168, *176–7*, 321, *321–2*
> (greater than) **166–7**, 169, *179–80*
< (less than) **166–7**, 169, *176–7*, 331
>= (greater than or equal to) **166–7**, 169, *179–80*
<= (less than or equal to) **166–7**, 169
{ } (braces) 69, 77, 167, 168,187, 189, 191
[] **246–8**, *258–60*, *264–5*
<> **251**, 253
// (comments) 68, *68*, 71–2, *76*, *76*
>> (input) *69*, *95–6*, 96, **96–8**, *100–1*, *140*, 318–22, *319–21*, *321–2*
<< (output) *69*, *94*, 95, **98–100**, *100–1*, *140*, 318–22, *319–21*, 321–2; *see also* output, formatted
& (reference parameter) **204–7**, *226*, 251, 253
\" 98
\' 98
&& (Boolean and) **165**, 166
|| (Boolean or) **165**, 166, 169, 170, *176–7*
! (Boolean not) **165**, 166
: 325
#include <cctype> **230**, *231*, *234*
#include <cmath> **209**, *211–12*, *229–30*
#include <cstdlib> **173**, *179–80*
#include <fstream> **318**, 318–22, *319–21*, *321–2*
#include <iomanip> **325**, *327–8*
#include <iostream> 94–5, **96**
#include <string> **96**, 122
#include <vector> **251**, *258–60*
#include (for class header files) 125, **134**, 139, *140*

abs (built-in function) 229
accessors *see* selectors
Adams, Douglas 36, 42, 43, 70
add.cc 301–2
addition 81, **83**
adventure games 30–1, 33
Algol 15
algorithms 14, 26
analysis (and design) 53–62
Analytical Engine 14
and (Boolean) **165**, *166*
animals.cc 100–1, 175, 176–7
append (method of string class) 228
argc 322–3, *323–4*
arguments *see* parameters
argv 322–3, *323–4*
arrays 8, **248–51**, 253–4, 255; *see also* sorting
assignment *76*, 78, **81–2**, 137, 192
assignment operator (of classes) 137
at (method of string class) 228, *234*, *236*, *237–8*
atan (built-in function) 229
attributes 5, 41, **42–4**, 46, 47–50, 51, 54, 120, 121
  defining 121–2,*138*
  identifying 57–8
averages.cc 312–13

B 16
Babbage, Charles 14, 352
base class 324–5
BASIC 15, 17, 37, 82
beta testing 113
BCPL 16
bigHole.cc 213–15
black box testing 301, 302
Blair, Tony 42
bool 43, **78**, 164; *see also* Boolean expressions; Boolean operators
Boolean conditions 7, 162–4, 167, 189
Boolean expressions 164
Boolean operators 164–6; *see also individual operators*
borrower.cc 156
borrower.h 155
boundary values (in test plan) 111–12, 114, 299

braces 69, 77, 167, 168,187, 189, 191
brackets (in expressions) 83, 89
break 171–2
bubblesort See *sorting*
bugs 14–15, 112–13
built-in functions **227–32**, *229–30*; *see also individual functions*
BURKS 353
Byron King, Lady Ada *see* Lovelace, Lady Ada

C 16, 17, 227, 245, 253, 325, 352
C# 16, 41, **358**
C++
  history 16, 46
  file extensions 22–3, 28
  popularity 7, 16
  system libraries 21, **25–6**, 95, 229
  Visual *see* Visual C++
c_str (method of stream class) 319, *319–21*, *321–2*
calculator.cc 197–8
calling program 134, 204
Cambridge University 16
case.cc 231–2
case study 270–95
cast 85, 88, *100–1*
ceil (built-in function) 229
cerr 96, 321, *321–2*, 323, *323–4*
char 43, **78**, 81, 98, 171
chocolate.cc 102–3
chocolate cake 13
chocolate digestive 1, 2
cin *69*, *95–6*, 96, **96–8**, *100–1*, *140*
classes 6, 119, 120, 272–9, 324–5; *see also* objects
  testing 302–8
  using 134–7
class interface 7, 119, 120, 125, 133–4. 153
close (method of stream class) 319, *319–21*, *321–2*
COBOL 15, 16
code reuse 41, 51, 54, 227, 271, 289
coding standards 68, 80
collections 49; *see also* arrays; vectors
command line arguments 322–4, *323–4*
comments 68, *68*, 71–2, *76*, *76*
comparison operators *see individual operators*
compilation 21, 22, **23–4**
  errors 24–5
compiler 21, 23, 24, 25, 28, 33, 354
concatenation (of strings) 81
conditions 7, 162–4, 167, 189
conditional statements 7, 17, **162–74**; *see also* if statement; switch statement
const
  with constants 85–6, 88, 91, *213–15*, 249, *261–2*, *262–4*, *264–5*
  in method prototypes 124, *138*, 203
  with parameters 251, 253
constant methods 124, *138*, 152
constant parameters 251, 253
constants 85–6, 88, 91, *213–15*, 249, *261–2*, *262–4*, *264–5*
constructor 124–5, 147, **150–1**
control variable
  in for loop 187–8
  in switch 170
Cooper, Doug 39
cos (built-in function) 229
cout *69*, *94*, 95, **98–100**, *100–1*, *140*
Croft, Lara 31

darkHole.cc 286–9
data hiding 7, 121,**153–4**; *see also* mutators; selectors
data types 43
  bool 43, **78**, 164; *see also* Boolean expressions; Boolean operators
  char 43, **78**, 81, 98, 171
  float 43, **78**, *101–2*, 326–9
  int 43, *69*, *76*, **78**, 81, *140*, 171
  string 43, *69*, **78**, 81, *95–6*, 98, 228–9, 246–8; *see also* reading strings
  void 45, **78**
dates (as attributes) 44–6, 170
debugger 25
debugging 112–13, 302
decrement 84
derived class 324–5; *see also* inheritance
determinate loop 7, 186, 188; *see also* for loop
Dev-C++ 354
DFE95 355
Difference Engine 14
division **83**, 84–5
do … while loop **191–2**,*197–8*, *213–15*
Downey, Allen 352
dragon, fire-breathing 31
driver programs 159, 233, 279, **303–8**, *304–5*, *306–8*
driving (compared to learning to program) 32–3
dry run of a program 302, 309–12

Eckel, Bruce 351–2
Elkner, Jeff 358
else clause **168–9**, *176–7*, *179–80*, *240–1*
else if clause **169–70**, *179–80*, *240–1*
endl *95*, 95, 99, *142–3*
entity-relationship modeling 42
eof (method of stream class) 319, *319–21*, *321–2*
executable file 23, 25–6, 28
exit **172–4**, *176–7*
expressions 81

fabs (built-in function) 229

`fail` (method of `stream` class) 319, *319–21*, *321–2*
famous people
  Adams, Douglas 36, 42, 43, 70
  Babbage, Charles 14, 352
  Blair, Tony 42
  Cooper, Doug 39
  Croft, Lara 31
  Downey, Allen 352
  Eckel, Bruce 351–2
  Elkner, Jeff 358
  Gates, William H. (Bill) 357, 358
  Gauld, Alan 358
  Gosling, James 357
  Hopper, Grace 352
  Kernighan, Brian 16, 352
  Lovelace, Lady Ada 14, 352
  Pythagoras 209, 276
  Richards, Martin 16
  Ritchie, Dennis 16, 352
  Spears, Britney 42
  Stroustrup, Bjarne 16, 351
  Tetley, Joshua 24
  Torvalds, Linus 353
  Turing, Alan 353
  van Rossum, Guido 358–9
  von Neumann, John 353
  Wilkes, Maurice 353
  Zuse, Konrad 15
file extensions 22–3, 25, 28, 120
files, for input and output 318–22, *319–21*, *321–2*
`find` (method of `string` class) 228
fine traditions (of programming books) 9, 94, 95, 254
`first.cc` 76
`float` 43, **78**, *101–2*, 326–9
`floor` (built-in function) 229
`for` loop 17, **186–9**, *194*, *240–1*, 255
formatted output 325–9, *327–8*
FORTRAN 15, 16
Free Software Foundation 354
functions **222–32**; *see also* built-in functions; non-member functions; overloading

g++ (compiler) 24, **354**
Gates, William H 357, 358
Gauld, Alan 358
GNU 354
golden rules of programming 65–70, **73**
Gosling, James 357
`grades.cc` 299–300
`greeting.cc` 95–6

header block 76, 176, *176–7*, 179, *179–80*
header file 6, **119–25**, 127–9, *138*, *302–4*
  file extension 22, 120
  format 121
`hello.cc` 94
help, getting 33, **34**, 35
Hopper, Grace 352
house style 68
human–computer interaction 93, 99–100, 168, 185–6, 190

IDE 23, 354–5
identifiers
  for attributes 121
  for constants 86
  for parameters 152
  for variables 77, 78, **79–80**
`if` statement 17, *69*, 163, **167–70**, *176–7*, *197–8*, *213–15*
`ifstream` 318–22, 318–22, *319–21*, *321–2*
implementation file 6, 119, 120, **147–54**
increment 82, **83–4**
indentation 68–71, 167, 168, 187, 189–90, 191, 194, *197–8*; *see also* programming style
indeterminate loop 7, 186, 189; *see also* `while` loop; `do … while` loop
infinite loop 190–1, 192
inheritance 324–5
initialising (variable) *see* variable initialization
input *69*, *95–6*, 96, **96–8**, *100–1*, *140*, 318–22, *319–21*, *321–2*
`int` 43, **78**, *101–2*, 326–9
integer division 84–5, 88–9, *102–3*
interactive development environments 23, 354–5
interface (of a class) 7, 119, 120, 125, 133–4, 153
`int main ()` 77, **223**, 322
`isalnum` (built-in function) 230–1
`isalpha` (built-in function) 230–1, *231*
`isdigit` (built-in function) 230–1, *231*
`islower` (built-in function) 230–1, *234*
`isspace` (built-in function) 230–1, *231*
`isupper` (built-in function) 230–1, *234*

Java 16, 41, **356–7**

kebab 9
Kernighan, Brian 16, 352

`layout.cc` 68–9
learned helplessness 35
learning environment 32
learning styles 31–2; *see also* study skills
learning to program 30–9
lectures, attendance 35
`length` (method of `string` class) 228, *258–60*
linker 21, **25–6**, 28, 95, 203, 225
Linux 353
Lisp 15, 16
literal values 81, 186, 203
Local Guide ix, xii, 22, 23, 24, 25, **27**, 28, 33, 70, 134–5, 149, 225, 229, 230

local variables 227
loops 7, **185–99**; *see also* determinate loop; indeterminate loop
Lovelace, Lady Ada 14, 352

Massachusetts Institute of Technology 16
methods 5, 41, 42, **44–5**, 46, 47–50, 51, 54, 120, **202–7**
  calling 135–6, *140*, *213–15*, 228
  constant 124, 152
  defining 122–4, *138*
  designing 59–60
  identifying 58–9
  implementing 121, **150–3**
  return type 45, 123, 148–9, 151
  return value 45, 136, 148–9
modulus 89
`money.cc` 101–2, 178, 179–80
money (data type for) 87
multiplication 83
mutators **152–3**, *155*, *156*

non-member functions 8, **223–7**, *226*, *234*
not (Boolean) **165**, 166
Notepad (text editor) 23

object code **25–6**, 28
object instances See *objects.*
objects 6, 15, **41–51**
  declaring 135
object types 42, 51, 120
  identifying 54–7
object-oriented programming 5, 16, 41, **46**, 51, 120, 133, 154
`ofstream` 318–22, 318–22, *319–21*, *321–2*
`open` (method of `stream` class) 319, *319–21*, *321–2*
operator precedence 83, 89
operators *see individual operators*; *see also* overloading
or (Boolean) **165**, 166, 169, 170, *176–7*
orange juice 2
ordinal data types 171
`output.cc` 327–8
output *69*, *94*, 95, **98–100**, *100–1*, *140*, 318–22, *319–21*, *321–2*
  formatted 325–9, *327–8*
`overload.cc` 330–1
overloading 329–34, *330–1*, *332*, *333–4*

parameters
  of a method 123, 152, 203; *see also* reference parameters; value parameters
  reference **204–7**, *226*, 251, 253
  value 203–4
Pascal 17, 39
`password.cc` 240–1
PFE (text editor) 23, 354
pie, custard (Bill Gates and) 357
Plankalkul 15
portability of programs 41, 76–7
`pow` (built-in function) 229
precedence (or operators) 83, 89
precision (of output) 326–9, *327–8*
private (part of a class) 120–1
programming languages
  Algol 15
  B 16
  BASIC 15, 17, 37, 82
  C 16, 17, 227, 245, 253, 325, 352
  C# 16, 41, **358**
  C++ See *C++*
  COBOL 15, 16
  evolution of 17, 19
  FORTRAN 15, 16
  history of 16
  Java 16, 41, **356–7**
  Lisp 15, 16
  Pascal 17, 39
  Plankalkul 15
  Python 17, 41, **358–9**
  Simula 15, 16
programming style 66–8, 70, 71–2, 80, 86, 99–100, 152, 169, 172, 191, 208; *see also* golden rules of programming; indentation
programs
  creating 22, 28
  design 53, **60–1**
  dry running 302
  executing 25, 28
  length of lines 70
  terminating 172–4; *see also* `exit`
  walkthrough 302
programs in text
  `add.cc` 301–2
  `animals.cc` 100–1, 175, 176–7
  `averages.cc` 312–13
  `bigHole.cc` 213–15
  `calculator.cc` 197–8
  `case.cc` 231–2
  `chocolate.cc` 102–3
  `darkHole.cc` 286–9
  `first.cc` 76
  `grades.cc` 299–300
  `greeting.cc` 95–6
  `hello.cc` 93
  `layout.cc` 68–9
  `money.cc` 101–2, 178, 179–80
  `output.cc` 327–8
  `overload.cc` 330–1
  `password.cc` 230–41
  `sheepDriver.cc` 304–5, 306–8
  `sheepOp.cc` 333–4
  `sheepValue.cc` 282–4
  `sortSheep.cc` 261–2, 262–4, 264–5, 319–20
  `sqrt.cc` 229–30
  `swap.cc` 226
  `tables.cc` 194

type.cc 231
username.cc 236, 237–8
validName.cc 234
valuableAnimal.cc 290–3
valuablePigs.cc 142–3
wanderingSheep.cc 140
wordCount.cc 321–2, 323–4
words.cc 258–60
wordStats.cc 309–10
prototype
of a function 223, 224, 329–30
of a method 122–3, 203
pseudo-code 59–60, 163–4, 271, 285
public (part of a class) 120–1
public interface 7, 119, 120, 125, 133–4, 153
push_back (method of vector class) 252, *258–60*, *262–4*
Pythagoras' theorem 209, 276
Python 17, 41, **358–9**

reading strings 97
recipes (as programs) 13–14
reference parameters **204–7**, *226*, 251, 253
requirements gathering 271–2
resetiosflags 327–9, *327–8*
return 151–2
return 0; 77
return type (of a method) 45, 123, 148–9, 151
return value (of a method) 45, 136, 148–9
Richards, Martin 16
Ritchie, Dennis 16, 352

selectors 147, **151–2**, 153, *155*, *156*
setiosflags 326–9, *327–8*
setprecision 326, *327–8*
setw 326–9, *327–8*
sheep.cc 211–12
sheep.h 138, 148, 210–11, 303–4, 332
sheepDriver.cc 304–5, 306–8
sheepOp.cc 333–4
sheepValue.cc 282–4
Simula 15, 16
sin (built-in function) 229
size (method of vector class) 252, *258–60*
software engineering 107–8
sorting **254–6**, 261–2, 262–4, 264–5, 282, *282–4*
sortSheep.cc 261–2, 262–4, 264–5, 319–20
source code 23, 25, 26, 28, 76
Spears, Britney 42
sqrt (built-in function) 209, 229, *230*, 276
streams 96
string 43, *69*, **78**, 81, *95–6*, 98, 228–9, 246–8
Stroustrup, Bjarne 16, 351
study skills 32, 35, 36
style 66–8, 70, 71–2, 80, 86, 99–100, 152, 169, 172, 191, 208; *see also* golden rules of programming; indentation
substr (method of string class) 228, *236*, *237–8*
subtraction 83
Sun Microsystems 357
swap.cc 226
swapping values 226, 256
switch statement 163, **170–2**, *197–8*, *213–15*, *240–1*
system libraries 21, **25–6**, 95, 229; *see also* linker
systems analysis 54
systems analyst 54, 62

tables.cc 194
test case 6, **109–10**, 112, 298
test data 109
test harness 303
testing **107–17**, 279, 297–309
black box 301, 302
boundary values 111–12, 114, 299
classes 302–8
example of 108–12
importance of 108, **113–14**, 298
not debugging 112–13
typical values 111–12, 114
by users 113
white box 301–2
test plan 6, **109–10**, 112, 298, **298–301**
Tetley, Joshua 24
text editors 23, 33
toast 2
tolower (built-in function) 231–2, *231–2*, 281
Torvalds, Linus 353
toupper (built-in function) 231–2, *231–2*, 247
transactions 58
truth tables 165
Turing, Alan 353
type.cc 231
typical values (in test plan) 111–12, 114

username.cc 236–7, 237–8
user testing 113
using namespace std; **76–7**, 122, 149

validName.cc 234
valuableAnimal.cc 290–3
valuablePigs.cc 142–3
value parameters 203–4
van Rossum, Guido 358–9
variable declarations *76*, 77, **80**
variable identifiers 77, 78, **79–80**
variable initialisation 82
variables 6, **78–9**, 91; *see also* variable declarations; variable identifiers

vectors 8, **251–3**, 254, 255, 256, *258–60*, *282–4*; *see also* sorting
`vi` (text editor) 23; *see also Vim*
Vim (text editor) 354
Visual C++ 23, 24
`void` (data type) 45, 123, 136
von Neumann, John 353

`wanderingSheep.cc` 140
`while` loop **189–91**, *197–8*, *213–15*, *236*
white space 97, 98
white box testing 301–2
Wilkes, Maurice 353
`wordCount.cc` 321–2, 323–4
`words.cc` 258–60
`wordStats.cc` 309–10

Y2K crisis 15

Zuse, Konrad 15

