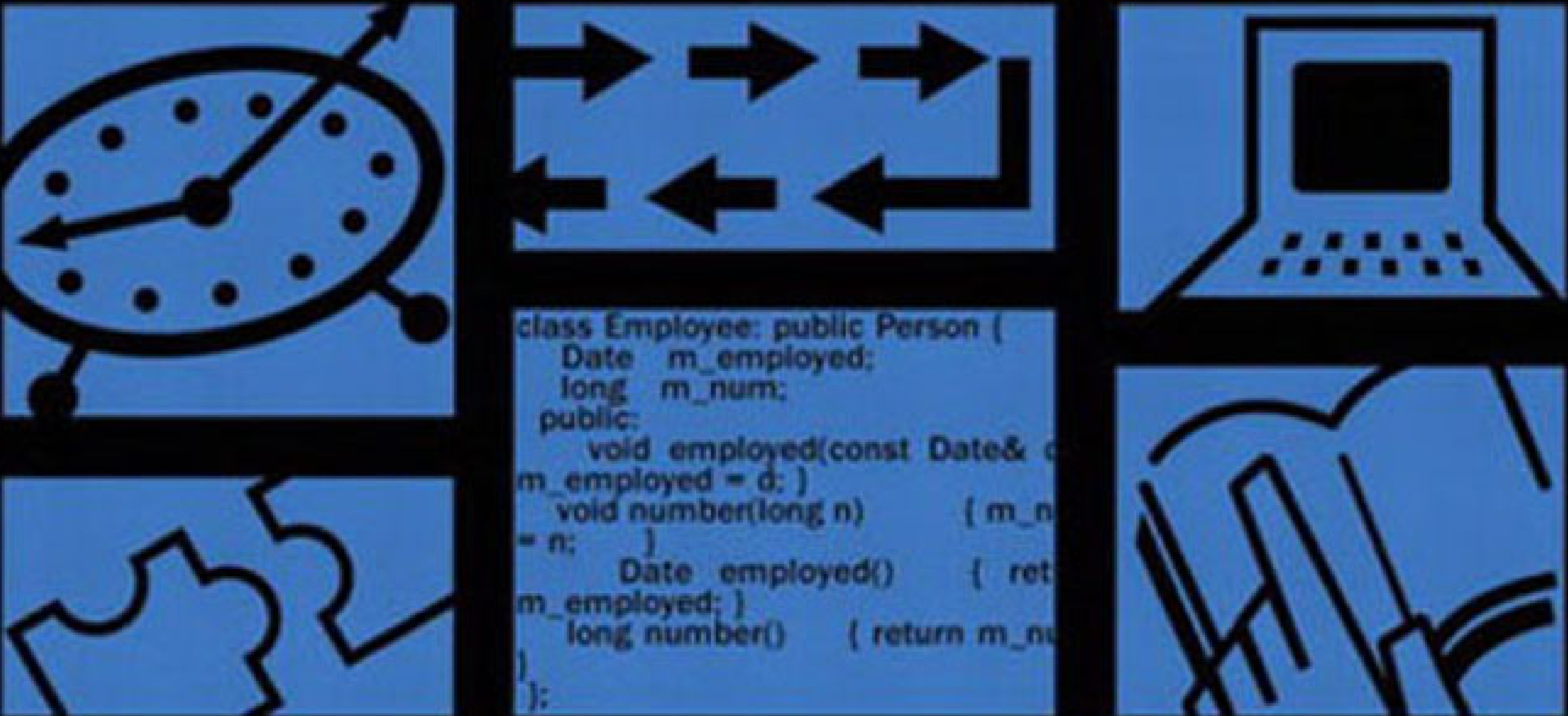
C++

BY EXAMPLE

UnderC Learning Edition



```
class Employee: public Person {
    Date    m_employed;
    long    m_num;
  public:
      void  employed(const Date& d
m_employed = d; }
    void number(long n)         { m_n
= n;      }
        Date   employed()      {   ret
m_employed; }
     long number()       { return m_nu
}
 };
```

Que®

Steve Donovan

C++ BY EXAMPLE

UnderC

LEARNING EDITION

QUE®
201 West 103rd Street
Indianapolis, Indiana 46290

Steve Donovan

C++ by Example: UnderC Learning Edition

International Standard Book Number: 0-7897-2676-9

Library of Congress Catalog Card Number: 2001096468

Printed in the United States of America

First Printing: December 2001

04 03 02 01 4 3 2 1

Trademarks

Warning and Disclaimer

Associate Publisher
Dean Miller

Acquisitions Editor
Todd Green

Development Editor
Laura Bulcher

Managing Editor
Thomas F. Hayes

Copy Editor
Kitty Jarrett

Indexer
D&G, Limited, LLC

Proofreader
D&G Limited, LLC

Technical Editor
Greg Perry

Team Coordinator
Cindy Teeters

Media Developer
Michael Hunt

Interior Designer
Karen Ruggles

Cover Designer
Radar Design

Layout Technician
D&G Limited, LLC

Contents at a Glance

Table of Contents

About the Author

Steve Donovan has been programming most of his life, mostly scientific and engineering applications. He did graduate work in nuclear physics and taught college physics for three years, which taught him the importance of language in learning science.

He has been with the Mining Research labs of CSIR South Africa for the past 10 years, primarily working with geophysical and geotechnical applications in Windows, in everything from Pascal to Assembler. C++ has been his tool of choice for five years, and he tries not to argue with Java and Visual Basic programmers.

Steve has developed the UnderC C++ interpreter to make life easier both for beginners and for experts who are tired of the compile-link-go cycle. Steve has released UnderC as open source, in the hope that someone else will debug it. He is not considered a gifted drawer of icons.

Dedication

This book is dedicated to Gill, who quietly supported me in my obsession, even when I didn't understand what I was doing. This book is also dedicated to my mother and sister, who thought it was all cool anyway, and my father, for his belief in the typewritten word and courage in the face of rejection slips.

Acknowledgments

We all learn at the feet of the masters (no matter how remotely), and I must thank Stanley Lippman, for his detailed discussion of how the machinery of C++ works, and Bjarne Stroustrup, for producing such an interesting language and patiently explaining the new C++ standard. I'd also like to thank Al Stevens, for his interesting column in *Dr. Dobb's Journal*.

Everybody has been very tolerant of my absence from ordinary life while I've been working on this project. I promise them that regular service will be resumed.

Tell Us What You Think!

As the reader of this book, *you* are our most important critic and commentator. We value your opinion and want to know what we're doing right, what we could do better, what areas you'd like to see us publish in, and any other words of wisdom you're willing to pass our way.

As an Associate Publisher for Que, I welcome your comments. You can fax, e-mail, or write me directly to let me know what you did or didn't like about this book—as well as what we can do to make our books stronger.

Please note that I cannot help you with technical problems related to the topic of this book, and that due to the high volume of mail I receive, I might not be able to reply to every message.

When you write, please be sure to include this book's title and author as well as your name and phone or fax number. I will carefully review your comments and share them with the author and editors who worked on the book.

Fax:	317-581-4666
E-mail:	`feedback@quepublishing.com`
Mail:	Dean Miller Que 201 West 103rd Street Indianapolis, IN 46290 USA

Introduction

Learning Languages from Books

Most people you speak to consider programming languages to be more difficult than human, or *natural,* languages. In many ways this is true. Programming is the art of preparing precise instructions for machines. These machines are very fast and obedient, but they are incredibly stupid, so everything has to be broken down into steps; nothing can be assumed. Also, most kinds of programming involve doing business or engineering calculations, so most people do not need to program. If setting a VCR feels like programming, somebody has not done his or her job because one of the goals of programming is to eliminate programming. But part of the trouble is that learning to program is often like learning Latin: It involves a lot of grammar and vocabulary, little practice, and no real-life applications. (Living languages, such as school French, can also be successfully killed by these problems.) Living human languages are best learned by speaking to people who know the language, and programming languages are best learned through *interactive conversation* with computers. Therefore, this book is a conversational course in C++.

Why learn C++? Despite the excitement surrounding Java, C++ remains the foremost industrial-strength programming language. It is incredibly adaptable. It is not the easiest language to learn and use, but you do not have to learn it all at once. Right from the start you can write (and read) interesting code that can do useful things.

By Example

People learn best by example. It's important to read as much good code as possible, just as reading lots of English is the only way to learn to write it. This book presents nontrivial applications of C++ that are interesting, and the case studies at the end of each chapter show C++ in use. I have provided the UnderC C++ interactive system for this edition. You should continuously play with the language, using UnderC, until you have enough confidence in the bricks to build your own houses. In addition, this book comes with the GNU Compiler Collection (GCC), which contains the most popular free C++ compiler.

The C++ Standard

C++ has been around for nearly 20 years , but it has only become an international standard in the past 5 years. Excellent free compilers are available that support that standard, so there has never been a better time to learn C++. Previously, each implementation of C++ had its own code libraries, so portability was a problem.

The strength of a language lies in its libraries. C++ has a powerful and elegant standard library that is available everywhere. Standard C++ is a high-level language that comes complete with the tools for most programming tasks.

The C Family of Languages: Some History

The history of C++ is interesting because it helps to understand where it comes from. On the one hand, there is a tradition of low-level power and efficiency that comes from C, and on the other hand it is a high-level object-oriented language.

C: Close to the Machine

The ancestor of C++ is C, which has been around since the early 1970s. It was developed by Dennis Ritchie at AT&T Bell Labs as a lean, fast language that is rich in low-level machine operations (such as bit twiddling) but can be moved (that is, *ported*) easily to other machines. As soon as UNIX was rewritten in C, it could itself be ported to many different architectures.

There is a certain cowboy spirit about C: You don't need to be too worried about data type safety if you know what you're doing. C is a very small language. It has no built-in I/O and file access or string manipulation capabilities, unlike BASIC, for example, which has `PRINT` and `INPUT`. In C, everything is done with libraries. C is often used to program devices as small as 12KB washing-machine controllers.

C++: "C with Classes"

C++ grew out of C in the early 1980s at Bell Labs. Bjarne Stroustrup was doing research in simulation, and he was inspired by the class concept of the Simula language. The original compiler translated C++ to C, but modern compilers go directly to machine code. The modern C++ language has grown throughout two decades from not having templates, exception handling, or multiple inheritance, to becoming a fine, stable adult.

C++ remains true to its C roots. It still relies on libraries for all its I/O functions and runs on small devices such as cell phones as well as on continentwide telephone networks. To this day, C++ remains incredibly backward compatible with C. A programmer can choose to get close to the machine or operate on a very high level; this flexibility is what gives C++ its special power.

Java: Universal Language

In the early 1990s, researchers at Sun Microsystems were looking at a reliable way to build the next generation of consumer devices. James Gosling and Patrick Naughton developed Java, which is syntactically similar to C++ (that is, it uses the same punctuation and many of the same keywords) but is designed to be a pure object-oriented language.

In consumer devices and Internet services, reliability, security, and portability are the key concepts. Therefore, Java omitted things like C++ pointers, which were error prone and open to abuse, and it generated an intermediate bytecode, which runs on a virtual machine. The Java Virtual Machine (JVM) also provides a "sandbox" in which small programs (called *applets*) can run safely without compromising security. There is a large library of code available for writing portable graphical user interface programs. Java programs can download new code dynamically (that is, as they are running).

Java is the property of Sun, which in 2001 won a court case against Microsoft for its modifications of Java, claiming that Microsoft was violating the terms of the Sun license.

C#: New Kid on the Block

In many ways, Microsoft's answer to Java is C#, pronounced "C sharp." Because Sun has effectively shut Microsoft out of more advanced Java versions, the company has been seeking a new platform for portable Internet services; this platform, which has recently been implemented, is called .NET. Microsoft has been thinking about this platform as long as Sun has; it was working on a similar concept, called Blackbird, at the same time that Java came out.

C# is a next-generation Java-like language that generates portable virtual machine code. Many people are asking whether we really need another programming language, but this splitting is probably inevitable because so many programming languages are owned by corporations, rather than being open standards, as C++ is.

UnderC: An Interactive C++

My goal in implementing UnderC was to produce a solid but interactive C++ system that would run a "pocket" version of the standard library. It is a half-compiler—source is compiled to intermediate stack code, which is executed immediately—and this is fast enough for most purposes. Programs can then be built by using compilers such as GCC if raw speed is required, and UnderC can link with dynamic link libraries (also known as shared libraries) that are compiled with any language.

You learn to program by interacting with computers, in the same way that you can learn French by conversing with a French person. Traditional compilers slow down this person–computer interaction by the compile-link-go cycle, in which the whole program is built and linked into an executable program; for even small programs, this cycle takes a second or two. An interpreter, on the other hand, brings up a program practically instantaneously. A magic number for humans is 300 milliseconds; anything more than this, and we start being conscious of a time lag. As programs get larger, the build cycle gets longer, and it gets harder to keep the conversation going, but because UnderC has no link phase, the conversation remains instant. Furthermore, a programmer can immediately test small parts of a system by using the interactive prompt; there is no need to write a 30-line program to test a single line. With UnderC, experimentation becomes less painful, and a kind of conversational flow develops, with no pause between typing and response.

How This Book Is Organized

The book is organized into two parts. Part I concentrates on C++ arithmetic expressions, program flow control, and functions—what is often called *structured programming*. In Part I you will learn how to do calculations and make decisions, and you will learn how to output the results. Structured programming is the art of organizing actions, using divide-and-conquer techniques to break complex operations into simpler ones.

Part II introduces *object-oriented programming*, which sees programs as consisting of objects that manage their own data and communicate with each other. In structured programming, there is a dictatorship of functions, and in object-oriented programming there is a democracy of objects. Neither programming approach, of course, is the full story. Some problems—for instance, those that handle simple tasks such as adding up numbers in a file—work best with structured programming.

Some experts would prefer to go straight to object-orientation and not worry about "obsolete" structured programming methods. In my experience with maintaining (bad) object-oriented programs, there is a lot of redundant code, mainly because the programmers never learned to organize their functions and separate out any common code. Also, to understand object-orientation you need to see how the traditional methods fail. Two case studies in this book tackle the same problem (drawing graphical shapes) from the two different perspectives to show the limitations of structured programming and how object-orientation can produce better programs.

Who Should Use This Book

This book does not assume any previous programming experience, although of course any exposure to other programming languages is very useful. Anybody wishing to seriously learn C++ will find everything they need to get started.

What the Reader Should Expect

All C++ language and library concepts in this book are illustrated with examples. There is a lot of code because it is important to see C++ in action. This book is a complete learning kit and contains a C++ interpreter (UnderC) for interactive exercises and two industrial-strength compilers (GCC and Borland) for building C++ programs.

What the Reader Should Not Expect

First of all, don't expect to master C++ in a few weeks. After all, no one expects a person to master a human language like Spanish immediately. But it will not take long to learn enough to write useful programs; it isn't necessary to master C++ to be a competent programmer. Mastery takes time and happens when you're using the language to do real things.

Second, this book is not a reference book. It is a tutorial introduction, and I did not want to confuse you with unnecessary detail. But as you continue to learn C++, you will need a good authority to consult. The classic is, of course, *The C++ Programming Language*, by Bjarne Stroustrup (3rd edition, Addison-Wesley, 1997), which discusses all features of the language in detail, with more than 300 pages devoted to the standard libraries alone.

Conventions Used in This Book

This book uses several common conventions to help teach C++. Here is a summary of those conventions:

EXAMPLE

Examples are indicated by the icon shown at the left of this sentence.

The typographical conventions used in this book include the following:

- Commands and computer output appear in a `monospaced computer font`.
- Words you type appear in a **`boldfaced computer font`**.
- *Italics* are used to introduce you to new terms.

In addition to typographical conventions, the following special elements are included to set off different types of information to make them easily recognizable:

NOTE
Special notes augment the material you read in each chapter. These notes clarify concepts and procedures.

TIP
Information that offers shortcuts and solutions to common problems is highlighted as a tip.

CAUTION
Cautions warn you about roadblocks that sometimes appear when working with C++. Reading the caution sections should help steer you away from trouble and wasted time.

Where to Find the Code

If you install the software on the accompanying CD-ROM, the example code for each chapter will be copied onto your hard drive. For instance, if you have installed to `c:\ccbx`, then Chapter 4's example code will be in `c:\ccbx\chap4`. As well as C++ source files, there are Quincy 2000 project files (`.prj`) for the case studies.

The CD-ROM will install the GNU C++ compiler GCC 2.95.2 and all the tools you need to do commmand-line and graphics programs in Windows. I recommend that you also get the Borland Free C++ compiler. Please see Appendix D, "Compiling C++ Programs and DLLs with GCC and BCC32" for details on where to find up-to-date versions of these compilers. If you are running Linux, your system probably already has the GNU C++ compiler. A command-line version of UnderC without graphics will be available for Linux early in March 2002; please consult the C++ By Example site listed below.

All updates, bug fixes, and general information can be found at `http://home.mweb.co.za/sd/sdonovan/ccbx.htm`.

Part I

C++ Fundamentals

Expressions and Variables

At heart, a computer is a machine for doing arithmetic and making decisions very fast, and a computer language is a precise way to give instructions to such machines. The simplest kind of C++ statement is an *expression*, which is similar to a mathematical expression, although a C++ expression is modified because it is hard to type square root signs and other mathematical symbols. I am going to introduce C++ expressions to you using the UnderC system; after you have installed the accompanying software, an UnderC icon will appear on your desktop. When you execute this program, a plain window will appear with a `;>` prompt. At this point, you can type any valid C++ statement, and it will immediately be evaluated.

In this chapter you will learn

- How to write common arithmetic expressions in C++
- How to declare and use variables
- The difference between floating-point and integer numbers
- How to manipulate text strings
- How to do input and output

Using C++ as a Calculator

When you fire up UnderC, you are presented with the command prompt `;>`. Now type **`2+3;`**—note the semicolon—and press Enter :

```
;> 2+3;
(int) 5
```

UnderC indicates input in black, output in green, and any errors in red; in this book we use bold for input and nonbold for output.

Most C++ statements end in a semicolon, so it's important to get used to adding a semicolon if you come from a language such as BASIC, which ends statements at the end of a line, without any terminating punctuation such as semicolons. C++ does not care about spaces, tabs, or new lines (which together are called *whitespace*). Take a look at the following example, where the user input is spread across several lines:

```
;>   2
 +
 3 -
1;
(int)4
```

As with all computer languages that ultimately derive from FORTRAN, multiplication in C++ must use the `*` operator:

```
;> 10*2 + 1;
(int)21
```

There are *operator precedence* rules that apply here, which are similar to the ones used in ordinary mathematics. Multiplication is understood to have a higher precedence than addition—that is, multiplication always happens before addition. If the precedence rules did not apply and the addition had been done first, the answer would have been 30 instead of 21! You can always insist on a certain order of evaluation by using parentheses:

```
;> 10*(2+1);
(int)30
```

Real, or *floating-point*, numbers can be used in expressions, as in the following example:

```
;> 2.3*2;
(double)4.6
;> 2*3 - 1.2;
(double) 4.8
;> 1/1.0e8;
(double) 1.e-008
```

Integers and real numbers are treated very differently from one another in computer arithmetic. Note that floating-point arithmetic dominates in an expression; if there are any floating-point numbers present in an expression, then those integers are converted into floating-point numbers. Very large or small numbers are expressed in *scientific notation*. In the preceding example, 1.0e8 is '1.0 times 10 to the power of 8'.

The usual functions of a scientific calculator are available. Most of these functions have the normal names, but, as listed in Appendix B, "A Short Library Reference," a few are written differently in C++ than in mathematics in general (for example, *arc sin* is written `asin()` in C++). Because a square root sign cannot be typed on a standard keyboard, you indicate a square root as `sqrt()`:

```
;> sqrt(2.3);
(double) 1.51658
;> sqrt(4);
(double)2.0
```

The mathematical functions like `sqrt()` expect a floating-point argument, so the integer 4 is implicitly converted to 4.0.

You can see that even at the very simplest level, C++ is useful as a calculator. I prefer to calculate by typing rather than by pressing buttons on a calculator. C++ does not provide specific financial functions, such as for compound interest; however, Chapter 2, "Functions and Control Statements," shows how straightforward it is to define new functions and use C++ as a customizable calculator.

Programmers used to other languages are often shocked to discover that there is no simple built-in way to raise a number to a power (exponentiation); C++ does not provide a `^` or `**` operator. In C++, as in C, this is done with the function `pow()`, which operates on floating-point numbers, as in the following example:

```
;> 1.0/sqrt(1+pow(3.2, 2));
(double) 0.298275
```

The big difference, so far, between C++ calculations and calculations on a scientific calculator, is that there is a distinction between integer and real numbers. The next section discusses this in more detail.

NOTE

As you go through this book, I encourage you to constantly try out things at the interactive prompt. It is easier to find out how something works (or whether it will work at all) in this way than to prepare a whole program and have to guess what went wrong when something does go wrong.

Numerical Types

All numbers are stored as a series of bits (short for *b*inary dig*its*) in memory. On most computers, the smallest addressable chunk of memory is a *byte*, which is 8 bits. The size, or *precision*, of a number is the number of bytes needed to fully represent it. On a 32-bit machine, the most natural number is 4 bytes, which makes up a *machine word*. Generally, a 32-bit machine moves data around most efficiently in 32-bit word-sized chunks. In older books, you will sometimes see a relic of the old DOS days, where *word* means 16 bits, and *double-word* means 32 bits, but when I use *word,* it generally mean 32 bits. In a few years, we will probably all be driving 64-bit machines, and then a word will be 64 bits; it is simply the most convenient and efficient chunk for a particular machine to process. You need to know these basic facts about machine arithmetic if you don't want to be unpleasantly surprised when you don't get the numbers you expect.

It is possible to use C++ to do infinite-precision arithmetic (or, more precisely, indefinite precision arithmetic), but that is a lot slower than normal machine arithmetic, and if you are working out the digits of pi, for example, the computation will never finish because pi cannot be represented by a finite number of digits.

Floating-Point Numbers

Two kinds of floating-point numbers are commonly supported in computers: single-precision and double-precision. A single-precision number uses one machine word, and a double-precision number uses two machine words.

On 32-bit machines, as a rule of thumb, single-precision gives you up to 6 valid digits, and double-precision gives you up to about 12 digits. C++ does all floating-point arithmetic in double-precision, but you can store your results in single-precision. It is a question of using a word when a word is enough; if you have tons of memory, then it is safer to always use double-precision.

Integers

There are four basic integer types: `char` (which is 1 byte), `short` (which is 2 bytes), `long` (which is 4 bytes), and `int` (which is a machine word in length; 4 bytes in the case of a 32-bit machine). The four integer types can be either signed or unsigned. For example, a signed `char` can store numbers from –127 to +127; an unsigned `char` can be from 0 to 256. Four byte integers go up to 2,147,483,647. A single ASCII character can be stored in a `char`, and this in fact is the chief use of the `char` type.

NOTE

You may recognize the four C++ integer types from Java, but beware! The C++ `char` type is *ASCII*, which is 8 bits; the Java `char` type is *Unicode*, which is 16 bits. Unicode encodes all the non-Roman characters (for example, Greek, Chinese) as well as ASCII. Standard C++ defines `wchar` (that is, wide `char`) for Unicode applications, and this is the same as the Java `char` type. Unlike Java, which is meant to be a machine-independent standard, C++ uses the most efficient sizes possible.

Variables

C++ would be of limited use if you could not use it to store the results of calculations, whether in memory or in a disk file. *Variables* allow you to give names to areas of memory for storing such results. Variables in C++ have definite types, such as `int`, `double`, etc., and more complex types that are built from these basic types.

Declarations

A *declaration* is a statement that sets aside some space in memory. It consists of a type, followed by a list of variables. A variable can contain a value, and you often set this value when you are declaring the variable. For example, the following declaration defines a variable `k` of type `int`, and it allocates 4 bytes for it:

```
;> int k  = 10;
```

After a variable has been defined, you can use it in any expression. For example, you can use the variable `k` as follows:

```
;> 2*k - 1;
(int) 19
```

If you don't declare a variable, it is considered an error. The system could detect such cases and implicitly declare a variable (as occurs in BASIC and many scripting languages), but this is a bad practice in real programs because it is too easy to misspell a name. Likewise, you cannot redeclare a variable, unless the variables are used in different contexts (which we will discuss in Chapter 2). UnderC tends to be more easygoing about variable redeclaration in interactive mode.

```
;> m;
CON 4:parse error
CON 4:Cannot find 'm'
```

You do not have to *initialize* the value in a declaration (that is, set the variable to some initial value):

```
;> int n;
;> n;
(int) n = 0
```

The value is `0` in this case, but the language does not guarantee that variables will be initially set to zero. You will see in Chapter 2 that in some common situations, declaring variables leaves their value completely undefined.

You can declare a number of variables in one statement. This statement declares six uninitialized variables:

```
;> int i1, i2, i3, alice, bonzo_the_clown, SinbadTheSailor;
```

There are rules for variable names: They may contain alphabetic characters, digits, and the underscore character (_). They can be lowercase or uppercase, but the system is case sensitive, so two variables `a` and `A` are considered distinct from one another.

CAUTION

Case-sensitivity in C++ is different from how most languages (including English!) do it so watch out for this. The name "Sinbad" is not the same as "SINBAD" in C++. The following declaration is completely legal, but it is silly because these are all distinct variables:

```
;>long onedog=1,Onedog=2,OneDog=3,ONEDOG=4;
```

There is a problem here because (a) these are different variables and (b) people usually won't read them as different variables. (Most other computer languages would regard these as the same variable; confusion is the result.) The best way to avoid this silliness is to have a *naming convention*. Most programmers prefer to make all variables all lowercase and use underscores (_) to separate words (for example, one_dog rather than onedog).

C++ supplies the operator `sizeof`, which you can use to determine how large (in bytes) a variable or type is:

EXAMPLE

```
;> float f;
;> double g;
;> sizeof(f);
(int) 4
;> sizeof(g);
(int) 8
;> sizeof(int);
(int) 4
```

Occasionally you will need to know the size of a type, such as `int`, in a program. Never assume that it is 4 bytes, and always use `sizeof(int)` to ensure that the program can still run on machines where `sizeof(int)` isn't 4. For example, on a SGI workstation an `int` is 8 bytes.

Assigning Values to Variables

After a variable is declared, you can put any value into it, provided that the value is of a compatible type. This is called *assignment*:

```
;> int n;
;> n = 42;
(int) 42
;> n = 2*n;
(int) 84
```

NOTE

A common problem is that people (quite naturally) confuse the operator = with mathematical equality. Since the first time you could grab a pencil, you have been brought up to read = as meaning "is equal to." It is indeed true that after the assignment in the preceding example, `n` is equal to 42. But = didn't mean they were equal before, because the variable's initial value was probably 0. And `n = 2*n` makes no sense whatsoever as a statement about equality. Other languages avoid this trouble by using an operator that won't be confused with equality (for example, `:=` in Pascal). A good habit to cultivate is to read `n = 42` as '`n` becomes 42' or 'assign 42 to `n`,' rather than '`n` equals 42'. The operator = does not compare two numbers; rather, it actively takes the right-hand side and puts it into the variable on the left-hand side, which is modified.

The variable can be used in a subsequent expression, and it will have the new value until the next assignment. An interesting fact is that the assignment statement `n = 42` is actually an expression: It has a value! This has two important consequences. First, you can say `m = n = 0`, because `n = 0` is an expression that has the value 0, and so `m` also becomes 0. (You may find it easier to read this statement as `m = (n = 0).`) Second, you can put assignments where other languages would never allow them. However, this is often a bad idea, and it can be a cause of much confusion.

We discussed earlier in the chapter that a variable has a given precision, which is the number of bytes it uses. It is safe to assign a numerical type that has a smaller precision to a variable. For example, you can assign an integer to a double variable; this process is called *promotion*. What happens if you try to assign a value that is larger than that variable can hold? This is very easy to show by using `unsigned char` because `char` can hold up to 255, and `unsigned` means the value will be displayed as both a character and as a decimal integer:

```
;> unsigned char ch;
;> ch = 32;
(unsigned char) ' ' (32)
;> ch = 300;
(unsigned char) ',' (44)
```

The value 32 is fine; it is the ASCII value of a space. But 300 is too big, and we get 300 – 256, which equals 44, which is a comma character. This is called *integer overflow,* and it happens with any integer type. Even 32-bit integers overflow eventually; if you keep a Windows 9x machine up for more than 49 days, the 32-bit clock counter (in milliseconds) overflows, and the system becomes unstable.

Constants: `const` and `enum`

It is very useful to define symbolic constants. The following example calculates the areas of circles, and so `PI` is defined as a constant:

NOTE

Constants are commonly written in uppercase so that it's easy to identify them in code.

```
;> double PI = 3.14;
;> double r = 2.3;
;> PI*r*r;
(double) 16.6106
```

Using `const` Declarations

Even though it is a constant, it is possible to assign values to `PI`, which seems to contradict the idea of a constant. (One of Murphy's Laws for software development is "Variables won't and constants aren't.") One C++ solution is the `const` modifier. When applied to a type in a declaration, it flags the declared symbol as not being modifiable:

```
;> const double PI = 3.14;
;> PI = 3;
CON 3: Cannot modify a const type
```

Constants, like variables, can be initialized with an expression, but that expression must be a compile-time constant. That is, it contains numbers (known as *constant literals*) or declared constants, as in the following example:

```
;> const double PI4 = PI/4.0;
```

In traditional systems, all of a program's statements are compiled into machine code, which is executed later. So constants cannot depend on the actual state of the running program. UnderC is more tolerant than traditional systems because in it, code is generated and executed immediately.

Just as with variable declarations, you can create several constants by using one statement, but you cannot declare a constant without immediately initializing it. The following statement creates three constants, `ONE`, `TWO`, and `THREE`. Note that you can define `TWO` in terms of `ONE`'s value, etc.—the value of a constant is set immediately:

```
;> const int ONE=1, TWO=2*ONE, THREE=3*TWO;
```

Using Enumerations

Another way to create constants in C++ is to use enumerations. By using the `enum` keyword, you specify a set of names and let the compiler generate unique values for them. The names are enclosed in braces ({}) and separated by commas.

EXAMPLE

```
;> enum { APPLE,BANANA,ORANGE,PEACH };
;> APPLE;  BANANA; ORANGE; PEACH;
(int) APPLE =  0
(int) BANANA = 1
(int) ORANGE = 2
(int) PEACH = 3
```

Using enumerations is particularly useful if you want a set of constants—such as `NORTH`, `SOUTH`, `EAST`, and `WEST`—but don't particularly care what their exact values are. You can still force the generation of particular values by explicitly setting a value; any following values will then default to that value plus one, as in the following example:

```
;> enum { LEFT = 1, RIGHT };
;> RIGHT;
(int) RIGHT = 2
```

It is even possible to fully specify these values:

```
;> const int n = 2;
;> enum { START = n,
    FINISH = 2*n
 };
```

Note that enumerated values, like other constants, can be constant expressions. Again, C++ does not care about whitespace, and you can organize the `enum` statement as you please. (But remember to end each with a semicolon; this is one of the two cases in C++ where a semicolon is needed after a brace.) In this case, many people think that it is better style to use a `const` declaration. Remember the purpose of constants and enumerations; they are to avoid putting mysterious numbers in programs. If I see the number '5000' in a program, it's puzzling, but if I see `MAXIMUM_OVERDRAFT` its meaning becomes self-explanatory (assuming that this was a banking program, of course) Also, when the bank changes its policy, I will not have to change every instance of '5000' scattered through several thousand lines of code. Well-named constants are an important part of documenting code.

Operators and Shortcuts

So far, you have met only the basic operators for addition, multiplication, and assignment. C++ has a very rich set of operators, including several designed to manipulate the individual bits of numbers directly and efficiently. I won't do those immediately, but concentrate on the most commonly used operators first. C++ came from C, which was designed by people who hated unnecessary typing, so you may find the operator symbols cryptic at first. But everyone gets used to them with practice.

Division and Remainder

In C++, division is expressed by the forward slash (`/`). The operator `/` has the same precedence as the operator `*`, just as in ordinary arithmetic, so the parentheses are necessary in the following example:

```
;> double x = 1.2;
;> (x+1)/(x-1) + 1;
(double) 12.
```

In mixed real/integer expressions, the integers are promoted into double-precision numbers. What if you want an integer result? If you are only interested in the integer part of a division operation, you can cast a double into an `int`; this expression `(int)x` is called a *typecast*. Going from a `double` to an `int` is called a *narrowing* conversion because the result has less precision. As you can see in the following example, it happens automatically when assigning a `double` to an `int`. Most compilers warn you about narrowing conversions, so it's best to explicitly use a typecast.

```
;> x = 2.3;
;> (double) 2.3
;> (int) x / 2;
(int) 1
;> n = x / 2;
(int) 1
```

It is common to need the remainder after an integer division. For instance, it will be zero if a divisor can divide exactly into a number. The `remainder` or `modulo` operator (`%`) does this for you. An interesting use of `%` is with the standard function `rand()`, which generates a large pseudo-random number. `rand() % 10` is guaranteed to be between 0 and 9:

```
;> 4 % 2;
(int) 0
;> 5 % 2;
(int) 1
;> rand() % 10;
(long int) 1
```

```
rand() % 10;
(long int) 7
;> rand() % 10;
(long int) 4
```

Logical Operations

As discussed earlier in the chapter the assignment operator (=) does not mean equality. C++ uses the `==` symbol to indicate equality. So you can read `a == b` as `a` equals `b`. Simularly, `!=` means ‘not equal to’.

```
;> n = 1;
(int) 1
;> n == 1;
(bool) true
;> n != 1;
(bool) false
```

The result of a comparison is of type `bool`, which is short for Boolean. (The founder of mathematical logic was named Boole.) This type was not part of the original C++ language, in which comparisons resulted in either one or zero. Generally, any nonzero result is considered true. The type `bool` is considered to be an integer, and the compiler converts it into a number if necessary.

There are other comparison operators as well: less than (<), greater than (>), less than or equal (<=), and greater than or equal (>=). These can be combined together with `&&` and `||`, which mean `and` and `or` respectively. You express `not` by using the operator `!` and `not equal` by using the operator `!=`.

EXAMPLE

```
;> 20 > 10 && n == 1;
(bool) true
;> n > 1 || n != 5;
(bool) true
;> n == 1+2 && 20 > 10 || n==5;
(bool) false
```

C++ notation takes some practice to get right. For example, note the respective precedence of `&&` and `||` in the third expression in the preceding example:

```
;> n == 1+2 && 20 > 10 || n==5;
```

`&&` is evaluated first, and then `||` is evaluated; the `+` operator has highest precedence of all. The Pascal equivalent would be expressed as follows:

```
((n = 1+2) and (20 > 10)) or (n = 5)
```

If you are unsure of the exact operator precedence in a C++ expression, there is no harm in adding extra parentheses. They have no effect on the resulting code, except that they make sure that it says exactly what you wanted.

Shortcuts

C++ inherited from C some operators (such as `++`, `—`, `+=`, `-=`, and `*=`) that increment and decrement variables directly, rather than forcing you to use expressions such as `n = n+1`. If `n` is an integer, then both `++n` and `n++` increment that integer; they differ in the value of the expression. `++n` is called the *prefix* increment operator, and `n++` is the *postfix* increment operator.

EXAMPLE

```
;> int nn = 1;
;> ++nn;
(int) 2
;> nn;
 (int) 2
;> nn++;
(int) 2
;> nn;
(int) 3
```

The prefix operator returns the value after the increment, and the postfix operator returns the value before the increment. There are also two decrement operators, `nn—` and `—nn`, which work similarly to `++n` and `n++`.

A number of assignment expressions apply an operation to the right-hand side of an equation. For instance, it is common to have statements such as `nn = nn + 2`. Although this is a valid C++ expression, in C++ this statement can be written as `nn += 2`. The operators `-=` and `*=` can be used similarly. The following code shows these shortcuts in use, assuming that `nn` was originally 2. Their value in each case is the value of `nn` after the operation.

```
;> nn += 2;
(int) 5
;> nn -= 2;
(int) 3
;> nn *= 4;
(int) 12
```

Generally, the shortcuts are not only easier to type, but produce faster code than writing them out in full (like `n=n+2`). This is part of the original C philosophy of being a fast language that's quick to type. The code `n=n+1` is not incorrect, but you will rarely see it in well-written C++ code.

Strings

So far we have discussed C++'s handling of both integer and floating-point arithmetic, and we have talked about how variables can be used to save the results of numerical expressions in memory. However, programs often have to manipulate text data. For instance, a compiler analyzes program text, and a word processor must detect the ends of words and of sentences. *String literals* such as "this is a string of characters" are part of the C++ language. However, C++ does not directly support character string data types. The standard C++ library has defined a type `string`, which is simple and intuitive to use.

The type `string`

The type `string` is not built into C++ like `int` or `float`, but it is made possible by the class mechanism of C++. Standard strings are *objects*. However, these strings are so straightforward to use that programmers can use strings as if they were built in, without worrying about the details. In the past few years, the standard C++ library has become widely accepted and implemented, and this means that there is agreement on how strings should work and how to manipulate them. String variables may be declared like any other type, as in the following example:

EXAMPLE

```
;> string s = "hello dolly";
;> s;
(string) s = 'hello dolly'
;> s.find("dolly");
(int) 6
;> s.find("swell");
(unsigned long int) 4294967295
```

`find()` is a special kind of function: It is associated with an object by using the dot (`.`) operator. An object has an associated set of functions called *methods*, which you can use to modify the object and ask for its properties. In the preceding example, the `find()` method is used to find the position of the *substring* "dolly", and it always returns the largest possible positive integer if it fails.

Concatenating Strings

A common operation on strings is to compose them out of smaller substrings. For example, the operator + has the effect of concatenating two strings:

EXAMPLE

```
;> string y = " you're so swell";
;> s + y;
(string) 'hello dolly you're so swell'
;> y + " dolly";
(string) 'you're so swell dolly'
;> s += " so fine";
(string) 'hello dolly so fine'
;> s.length();
(int) 20
```

Similarly, the operator += appends a string to another string, modifying the target string. There is no – operator because there is no C++ operation that is like subtraction on strings. At any point, you can find out the length of the string by using the method `length()`, which takes no arguments. In such a case, you put an empty list after the name.

Note that the first example uses a single quote character within the double quotes: " you're so swell".

How would you put a double quote in a string literal without prematurely ending the string? C++ allows you to embed special characters in a string, by using the backslash escape character (`\`) followed by a character. Note that you must double up the backslashes if you want to do a DOS-style pathname; otherwise, each `\` in the pathname will be interpreted as an escape character. Other very useful escapes are `\n` (for return) and `\t` (for tab). In the following code, I have created three strings containing special characters. The last is printed out on three lines because it contains two return characters:

EXAMPLE

```
;> string path = "here is a \"quote\"";
;> path;
(string&) 'here is a "quote"'
;> path = "c:\\programs\\ucw";
(string&) 'c:\programs\ucw'
;> path = "line1\nline2\nline3";
(string&) 'line1
line2
line3'
```

Finding and Extracting Substrings

The `substr()` method takes two arguments; the first is a start position, measured from zero, and the second is a character count. The `substr()` method copies the specified number of characters from the start position, and it is like the Borland Pascal `copy` function or the BASIC `left$` function. In the example, I use `substr()` to extract substrings. The third call uses the index returned by `find()`:

EXAMPLE

```
;> p = "hello dolly";
(string&) 'hello dolly'
;> p.substr(0,1);
(string) 'h'
;> p.substr(1,2);
(string) 'el'
;> p.substr(p.find("dolly"), 5);
(string) 'dolly'
```

As mentioned earlier in the chapter, the string's `find()` method looks for the first occurrence of a substring. It returns an index that `substr()` can use (that is, it is not negative). Generally, you should always check the return value before using it.

The `replace()` method takes a specified substring, like `substr()`, but it replaces the substring with the given string, rather than extracting it:

```
p.replace(0,5,"goodbye");
(string&) 'goodbye dolly'
```

Input and Output

To be useful, a program must communicate with the world and save calculated results on disk. Most C++ systems do not automatically show the result of every expression the way UnderC does. C++ programs use the *iostreams* library, which defines input and output "streams".

Writing to `cout`

With C++, a program can direct values and text to output by using the *insertion* operator (<<), acting on `cout` (pronounced "see-out"), which is a special variable that represents the standard output stream. (Bjarne Stroustrup suggests we call << "put to.") The following example uses << to output a string literal and an integer. Note that the items follow each other on output with no extra spaces:

```
;> cout << "nn = " << nn << endl;
nn = 2
(ostream&) ostream {}
```

Although this is an expression that has a displayed value, we are not interested in the result. The statement is executed for the effect of the insertions, which is to output a text value, an integer value, and `endl`, which forces a new line. The following example shows the effect of `endl`, and note that in this example, the last integer has not been left on its own line:

```
;> cout << 1 << endl << 2 << endl << 3;
1
2
3(ostream&) ostream {}
```

Evaluating an expression because of its effect on the environment or some object is common in programming. We are not always interested in its value.

The operator << has a lower precedence than any other mathematical operator, so you can safely use expressions with it, but comparisons are a problem. You usually get some sort of error message when you use it, so you will know when you have a problem. The first statement executes fine, because of <<'s low precedence, but the second causes an error:

```
;> cout << "the answer is " << 20*10+1 << endl;
the answer is 201
;> cout << "a < b = " << a < b << endl;
CON 36:Could not match void operator<<(int,_Endl_);
```

Because < has lower precedence than <<, the system has to make sense of (`b << endl`), and it fails. This is another instance where you might want to use extra parentheses to clarify the precedence order you desire.

Reading from `cin`

You can use the extraction operator (>>), called "get from," to retrieve values from the standard input stream, `cin` (pronounced "see-in"). This code prompts the user for an integer; in the case of the example that follows, **`42`** is then typed in:

```
;> cin >> nn;
42
(istream&) istream {}
 ;> nn;
(int) nn = 42
;> int i1,i2;
;> string s1,s2;
;> cin >> i1 >> i2 >> s1 >> s2;
10 20
30 dog
(istream&) istream {}
```

You can extract multiple values from the input stream, ignoring whitespace in the input. Note that strings are assumed to be separated by spaces; the input is assumed to form a continuous stream of characters, not be broken by lines.

Other languages, such as BASIC, have line-oriented input (that is, each input statement grabs a whole line), which makes it quite difficult to perform stunts like the previous example. However, sometimes in C++ you need to read in a whole line at a time. You can do this using the `getline()` function, which works with an input stream such as `cin` and reads in the line as a string:

```
;> getline(cin,s1);
a sentence is composed of words
;> s1;
(string) 'a sentence is composed of words'
```

Writing and Reading Files

Writing a file involves creating an output stream and using the insertion operator, exactly as for standard output. First you declare a variable of type `ofstream`, and then you use it to open the file. Then the file is written to in precisely the same way as writing to `cout`, and when you are finished writing to the file, you close it:

EXAMPLE

```
;> ofstream out;
;> out.open("tmp.txt");
(bool) true
;> out << i1 << " " << i2 << endl;
(ostream&) ostream {}
;> out.close();
```

You view the new file by using Notepad, or you can directly execute the DOS `type` command from the `;>` prompt:

```
;> #x type tmp.txt
10 20
```

To read a file, you need to use `ifstream`, which is a file input stream. Again, the sequence of operations is the same; you open the stream, giving it the name of an existing file, and then use `>>` to extract values from the stream. When finished, it's important to close the file.

EXAMPLE

```
>ifstream in;
;> in.open("tmp.txt");
(bool) true;
;> in >> i1 >> i2;
;> in.close();
```

Reading from Strings

It is often useful to extract numbers from a string. The easiest way to do this is by using a special kind of input stream, called `istringstream`:

EXAMPLE

```
;> p = "10 20 fred";
;> istringstream is(p);
;> is >> i1 >> i2 >> s;
(istream&) istream {}
;> cout << i1 << "," << i2 << s << endl;
10,20fred
(ostream&) ostream {}
```

What's Next

I have shown you how C++ can be used as a sophisticated calculator with the UnderC interactive system. Arithmetic expressions are written using variables and constants, combined together with operators such as * and +. The main difference from a normal scientific calculator is that integer and real numbers are treated differently. You can also manipulate character strings and do input/output. Next, I will show you how to extend the calculator by writing your own functions and how to control the execution of statements with loops and decisions.

2

Functions and Control Statements

In Chapter 1, “Expressions and Variables,” you learned about C++ expressions and the most important operators used in expressions. In addition, you learned how to manipulate character strings, display values, and write to disk files. So far we have not been programming. Programming involves making decisions and repeating operations; the building blocks of programs are *functions* and *control statements*. A C++ program consists of one or more files that contain *statements* (which are its sentences) organized into *functions* (which are the equivalent of paragraphs). For instance, a program to calculate the average value of some numbers would have to repeatedly get a value, add it to the sum, until there were no more values. And finally it would divide that sum by the number of values and print out the result. If that result was a student’s final aggregate mark, then the program may print out a message if that mark is below some value. I could give this description to a person, and they would be able to sit down with a calculator and process some test results. Computers are not intelligent enough to use English, which is in any case far too vague and wordy. So programming C++ involves precisely specifying each step (for example, what does ‘no more values’ mean?) so there is no room for misinterpretation.

In this chapter you will learn

- How to define functions
- How to execute statements conditionally
- How to repeat statements
- The difference between global and local variables

Defining Your Own Functions

Functions are collections of C++ statements, which are usually fairly short and do some specific thing. These are the two main reasons why functions are essential in programming. I may need hundreds of lines to specify a complex task, and that is simply too long for most humans to follow. It's easy to get lost in big programs, just as in big documents, and you can't assume that your reader has the time or patience to work it out. Dividing code up into functions is like dividing text into named paragraphs and chapters. Programming is like cooking—complicated recipes are broken down into sub-recipes, like making the sauce, cooking the vegetables, etc. The built-in functions like `sqrt()` are basic steps that all cooks should know. It is now time to write your own recipes.

Squaring Numbers

You may have found it odd that there is no built-in way to do something as simple as square a number in C++; there is no `n ^ 2` as in BASIC for this common operation. You can write `n*n`, but squaring expressions gives inefficient code, such as `(n+1)*(n+1)`, which might be good mathematics, but involves repeating the addition, looks clumsy, and can cause errors. In C++, however, it is straightforward to define a function that squares an integer:

EXAMPLE

```
;> int isqr (int x) { return x*x; }
;> isqr(2);
(int) 4
;> isqr(2*3) - 1;
(int) 35
;> double cube (double x) { return x*x*x; }
;> cube(2);
(double) 8.
```

The function `isqr()` is now available for use in any expression, just like a built-in function such as `sqrt()`. `isqr()` takes an argument of type `int` and returns a value of type `int`. `isqr(2)` is called a *function call*; we speak of `isqr()` being passed an argument of type `int`. You can likewise define `cube()` as a function that returns `double` and is passed a double value. Note that, just as with built-in functions, integer arguments are automatically promoted to `double`.

You can start customizing C++ to be a financial calculator by defining a compound interest function that compounds the interest annually. If you need a function that compounds the interest every quarter, then it can be defined in terms of the `compound_annually()` function:

EXAMPLE

```
;> double compound_annually(double cap, double interest, int nyears)
;1> {
;1>  return cap*pow(1.0 + interest,nyears);
;1> }
;> double compound_quarterly(double cap, double interest, int nyears)
;> {
;>     return compound_annually(cap,interest/4.0,4.0*nyears);
;> }
```

Argument names should be meaningful—don't call the interest argument just 'i' and the number of years just 'n'. These function names may be too tedious to type for a quick calculation, but it's possible to define a C++ *macro* to use as a shorthand, as in the following example:

```
;> #define CA compound_annually
;> compound_annually(1000,0.05,10);
(double) 1628.89
;> CA(1000,0.06,10);
(double) 1790.85
;> compound_quarterly(1000,0.06,10);
(double) 1814.02
;>
```

The Anatomy of a Function

A function definition looks rather like a declaration: It includes a type (that is, the return type) and the function name, followed by the formal argument list and the actual code, which is enclosed in braces (`{}`) and must contain a return statement. The argument list consists of declarations separated by commas. Sometimes people speak of a function's *parameters*, which are synonymous with *arguments*. A function can call other functions; it can have no arguments, or it can have many; and whitespace is not significant. The following example defines a function `pi()`, which just returns a constant value and a function `circle_area()`, which uses `pi()` and `sqr()` to calculate the area of a circle. The function `add()` takes two integer arguments.

EXAMPLE

```
;> double pi() {
;1>  return 3.1412;
;1> }
;> double sqr(double x)  { return x*x; }
;> double circle_area(double r) { return pi()*sqr(x); }
;> int add(int a, int b)
;> { return a + b; }
;> add(10,20);
(int) 30
```

The idea of a function not taking arguments is strange because such a beast does not fit easily into mathematics. The function `pi()` in this example might seem to be a silly way to define a constant, but such functions have interesting uses. The function with two arguments `int add(int a, int b)` shows each argument being separately declared. I mention this because it's natural to think you could declare that function more easily as `int add(int a,b)`, which is how ordinary declarations are written (especially since some languages like Pascal do that!) But that is not how the C++ language works.

Functions That Don't Return Values

Sometimes you aren't interested in a return value, but everything in C++ must have a type. The type `void` indicates "no type" or "I don't care" when used as the return type. The following function prints out an integer value, but does not return a value.

```
;> void show(int I)
;> {
;1> cout << "I = " << I << endl;
;1> }
;> show(42);
I = 42
;>
```

There are two interesting things about this function: It has no return statement, and it genuinely has no value, which you can see when you use it interactively at the UnderC prompt. The `void` function is the only kind of function that doesn't need a return statement—because there isn't anything to return! Not returning a value is again another odd thing mathematically, but C++ functions do not play by the same rules as mathematical functions. Other languages call `void` functions *procedures* (for example, Pascal) and subroutines (for example, BASIC). Why would you need functions that don't return values? You might want to collect commonly used statements together. For example, the following example produces a little report:

EXAMPLE

```
;> void interest_report(double capital, double interest_rate) {
;1>  cout << "Original capital was " << capital << endl;
;1>  double interest = capital*interest_rate;
;1>  cout << "Interest was " << interest << endl;
;1>  cout << "New capital is " << capital + interest << endl;
;1> }
;> interest_report(1000,0.06);
Original capital was 1000.000000
Interest was 60.000000
New capital is 1060.000000
;>
```

Side Effects of Functions

The function `interest_report()` is only used for its effect, which is to print out some stuff. And this is why we made it a `void` function. But, consider the following variable and a function that modifies it:

```
;> int kount = 0;
;> int isqr(int I) {
;1> kount++;
;1> return I*I;
;1> }
;> isqr(2);
(int) 4
;> kount;
(int) kount = 1
```

The main purpose of this function is to square integers, but it also increments a count. You would use this function to get an idea of how many times integers are squared in a program. This secondary effect of a function is called a *side effect* and can be a recipe for disaster if you aren't careful. A good rule is for each function to do one thing only. If a function is supposed to calculate a tax rate, then it shouldn't moonlight and update the database as well. If you can't come up with a clear name for a function, then its role is probably too vague and needs to be clarified further.

Control Statements

Until this point in the book, statements have been executed in sequence. The flow of control in a C++ program is linear; it may pass to a function and back, but it always passes through each statement. Often you need to make decisions and you need to repeat actions. These are the basic reasons for control statements.

The `if-else` Statement

Programs often need to make decisions. For example, if a student scores less than a certain required minimum, action has to be taken; if an account is overdrawn more than a prescribed number of days, bills need to be sent out and interest rates raised. The C++ `if-else` statement handles these types of decisions:

```
;> int mark = 49;
;> if (mark >= 50) cout << "pass\n";
;> else cout << "fail\n";
fail
;>
```

This statement consists of three parts:

- A *condition*, which can be any Boolean (or integer) expression enclosed in parentheses
- A statement that is executed if the condition is true
- An (optional) `else`, followed by a statement that is executed if the condition is false

Pascal programmers should note that there is always a semicolon after the `if` part. Not every `if` statement has an `else`, as in the following example:

```
;> const double MAX_OVERDRAFT = 5000;
;> void check_account (double overdraft) {
;1>  cout << "Overdraft is " << overdraft;
;1>  if (overdraft > MAX_OVERDRAFT)
;1>     cout << " overdrawn by " << overdraft - MAX_OVERDRAFT;
;1>  cout << endl;
;1> }
;> check_account(4000);
Overdraft is 4000.
;> check_account(6000);
Overdrarft is 6000. overdrawn by 1000.
```

The warning about the overdraft happens only if it is excessive. (Incidentally, note in this example how you can split up a line of output over several statements. Output streams, like input streams, are not line based, which is why you need `endl`.)

Generally speaking, you can put any valid C++ statement after `if`. The statements controlled by the `if-else` statement can themselves be `if-else` statements. This is a very common pattern, which is commonly used where there are multiple choices to be made. Save the following as `mult-if.cpp in Notepad`, and load it using the `#l` command. You can then test it as before:

```
// mult-if.cpp
string test_if (int mark)
{
  string symbol;
  if (mark < 50) symbol = "F";
  else if (mark >= 50 && mark < 55) symbol = "E";
  else if (mark >= 55 && mark < 60) symbol = "D";
  else if (mark >= 60 && mark < 65) symbol = "C";
  return symbol;
}
;> #l mult-if.cpp
;> test_if(55);
(string) 'D'
```

Experimenting with `if` statements with UnderC reveals curious behavior. Say you type an `if` statement that should execute, but nothing happens until you type another statement. This happens because the compiler is looking ahead to see whether an `else` is forthcoming. It's easy to force this by adding an extra semicolon, creating an *empty statement*, as in the following example:

```
;> if (20 > 10) cout << "yeah!\n"; ;
yeah!
```

CAUTION

The condition of an `if` statement might be just about any C++ expression that could possibly make sense as a `bool` or an integer. Java insists that it must be a `bool`, which is a good attitude to take in C++ as well. Remember that the assignment operator returns a value, which means that the following code is valid C++: `n = 1` assigns 1 to `n`, and because the value 1 of that expression is nonzero, the condition is true. It is very easy to confuse == with =! Many compilers (like BCC55) will give you a warning about 'possibly incorrect assignment'.

```
;> int n = 2;
;> if (n = 1) cout << "was one!\n"; ;
was one!
```

Blocks

You have previously encountered *blocks* in function definitions; the statements between braces (`{}`) form a block. Blocks are used to collect several statements together, and they may themselves contain blocks. Blocks are often controlled by `if` statements, as in the following example:

EXAMPLE

```
;> double a = 1, c = 0;
;> if (a > 0) {
;1>  double b = 2*a;
;1>  cout << "b is " << b << endl;
;1>  c = a - 1;
;1> }
 b is 2
;> b;
CON 3: Parse Error
CON 3: Cannot find 'b'
```

The interpreter prompt changes to show how deep the block level is at the moment. A declaration is a statement, so blocks can contain declarations, but these declarations are private to that block.

The `while` and `do-while` Statements

Programming any language becomes much more entertaining when statements can be repeatedly executed in a loop. *Loop* means that the flow of control jumps (or loops) back while some condition is true, as in the following example, which demonstrates the `while` statement. A `while` statement is similar to an `if` statement; there is a condition in parentheses followed by a *controlled statement*. The controlled statement is executed if the condition is true but will **continue** to be executed until that condition is false. The `while` statement followed by the controlled statement is often called a `while` loop:

```
;> int k = 0;
;> while (k < 10) k = k + 1;
;> k;
(int) k = 10
```

The controlled statement `k = k + 1` executes 10 times, until the condition fails, because `k` was no longer less than 10. Incrementing a variable like this isn't good style in C or C++, however. Let's watch this loop in action again, this time using the postfix increment operator `++` (which returns the value of `k` before it is incremented):

```
;> k = 0;
;> while (k < 4) cout << k++ << endl;
0
1
2
3
;> k;
(int) k = 4
```

The `while` statement can control any kind of statement, but it is especially used with blocks. The `while` statement is not executed if the condition is false at the beginning. There is another form of `while` loop in which the statement is always executed at least once: the `do-while` loop. The following function reads and sums numbers from a file until the end of the file:

```
int sum_file(string file)
 {
  ifstream in;
  in.open(file.c_str());
  int val=0,sum = 0;
  do {
    sum += val;
    in >> val;
  }  while (! in.eof());
  in.close();
  return sum;
 }
```

NOTE

The `eof()` method usually returns false, so `! in.eof()` is true.

The problem with using a `while` loop here is that we must not use the value if the EOF becomes true, which means we have gone beyond the EOF, and the value is not meaningful (it is usually the last value read, but that's accidental). Replace the `do-while` with the following loop and note that it appears to count the last number twice:

```
while (! in.eof()) {
  in >> val;        // get the value
  sum += val;       // could be EOF at this point!
}
```

A much more convenient method is to use the result of the read operation itself, which is nonzero unless it is past the end of the file:

```
while (in >> val)
   sum += val;
```

The `for` Statement

It is common to want to use some statements a certain number of times or take a variable (that is, a *loop counter*) from a start to a finish value. The `while` statement is not ideal for this job because you must initialize the variable separately and remember to increment it each time, both of which are tasks that you can easily get wrong. The initial value of the variable may be anything, and if you don't increment it, the loop happily goes on forever—it is an *endless loop*—because the condition is always true. The `for` statement is designed to make these kinds of loops easier and less error-prone. It has three parts, separated by semicolons: an initialization, a condition, and an operation that happens on each iteration. The following is the simplest form of the `for` statement, and you will learn to type it in your sleep:

EXAMPLE

```
;> for(k = 0; k < 4; k++) cout << k << endl;
0
1
2
3
;>
;> k = 0;
(int) 0
;> while (k < 4) {
;1> cout << k << endl;
;1> k++;
;1> }
```

```
0
1
2
3
```

The first part of this `for` statement is executed once in the beginning; the second part is executed before the controlled statement; and the third part is executed after the controlled statement. Like a `while` statement, the `for` statement is not executed if the condition is initially false. In this example, the `for` loop is completely equivalent to the `while` loop, which is more complicated (because it needs a block). With a `while` statement, it is particularly easy to forget to increment the counter at the end, especially if there are many statements in the block.

The `for` statement does not trip off the tongue (or the fingers) as easily as BASIC's `for` loop. It is the ugliest way to get from 0 to `n-1`, but it is very flexible. It automatically goes from 0 to `n-1`, which is the way C++ prefers to count. You can declare variables in the initialization part. The following example sums the integers from 0 to 9:

EXAMPLE

```
;> int sum = 0;
;> for(int j = 0; j < 10; j++) sum += j;
;> sum;
(int) 45
;> j;
CON 6: Cannot find 'j'
```

In this example, note the shortcut addition operator. Any variables (here `j`) declared in the initialization part are visible only within the loop. So we get an error trying to access j outside the loop. The third section of a `for` statement is usually a plain increment (like j++) but it can be any expression. A `for` loop can go backward or go forward with a skip value, as in the following example:

EXAMPLE

```
;> for(k=3; k > 0; k—) cout << k << endl;
3
2
1
;> sum = 0;  // this will sum the even numbers up to (and including) 20
;> for (k=0; k <= 20; k+=2)  sum += k;
```

The `for` statement can control any statement, including another `for` statement. Therefore, you can use `for` to write nested loops, as in the following example:

EXAMPLE

```
;> for(int i = 0; i < 3; i++)
;1} for(int j = 0; j < 3; j++)
;2}   if (i > j) cout << '(' << i << ',' << j << ")\n";;
(1,0)
```

```
(2,0)
(2,1)
```

The `loop` variable does not have to be an integer, unlike in most other languages. In the following example, you vary a `float` variable and print out a small table of trigonometric functions. The shorthand assignment-with-addition operator (`+=`) is very convenient for incrementing `x` by a small amount:

```
for(float x = 0; x < 1.0; x += 0.1)
  cout << x << ' ' << sin(x) << endl;
0.000000 0.000000
0.100000 0.099833
0.200000 0.198669
0.300000 0.295520
0.400000 0.389418
0.500000 0.479426
0.600000 0.564642
0.700000 0.644218
0.800000 0.717356
0.900000 0.783327
1.000000 0.841471
```

The C++ `for` statement takes some getting used to, but it is very powerful for expressing many different kinds of loops.

The `switch` Statement

It is common to have a number of actions that are controlled by a set of values. Because it can be clumsy to use multiple `if-else` statements, C++ offers the `switch` statement. Within a `switch` block there are *case labels* for each value, followed by statements that will be executed. These are usually followed by a break statement, which explicitly forces the flow of control out of the block. There is a special-case label called `default`, which acts rather like an `else` statement.

```
void suburb(int prefix)
{
  switch(prefix) {
  case 728:
    cout << "Melville";
    break;
  case 482:
    cout << "Parktown";
    break;
  case 648:
  case 487:
    cout << "Yeoville";
    break;
  default:
    cout << "*Unrecognized*";
    break;
  }
  cout << endl;
}
```

This example shows how you handle cases where the action is triggered by more than one value. The flow of execution jumps to the case label, and then it falls through to the next label. If you don't put in a break, execution keeps going through every statement. In the following example, if l is 1 then **both** cases are executed.

```
;> switch(1) {
;>  case 1:  cout << "one ";
;>  case 2:  cout << "two ";
;> }
one two
```

You need to be careful with `switch` statements because they do not work like equivalent statements in other languages. At first, you may find it easy to forget the `break`. Even experienced programmers get their breaks mixed up; one of the most expensive bugs ever took out millions of telephones in the eastern United States, and it was due to a misplaced `break`.

Scope

Earlier I remarked that variables declared within blocks are not defined outside that block. It would be hard to do serious programming if this were not true; you would be forced to come up with a unique name for every variable. The part of a program in which a variable is visible is called that variable's *scope*.

Global Variables

Variables declared outside functions are called *global* because they are visible from any function in the system; such a variable has global scope. Side effects are a problem in large programs, where it is hard to know exactly where a variable was changed. Unlike in some languages, C++ global variables are automatically available to all functions defined in a file (this is called *file scope*).

Local Variables

Declarations within a block—that is, *local variables*—are visible only inside that block. Local variables have local scope and will hide any variables that are declared outside the block. That is, the local definition always dominates. The formal arguments (or parameters) of the function are considered to be declared as local variables. In the following example, the variable `var` is redefined within `scope_test()`.

EXAMPLE

```
;> int var = 1;
;> void scope_test(int a, int b) {
;1>  int var = a + b;
;1>  cout << a << " " << b << " var = " << var << endl;
;1>  a = 999;
;1> }
;> int a = 11;
;> scope_test(a,22);
11 22 var = 33
;> var;
(int) 1
;> a;
(int) 11
```

In this example, there is a variable `var`, which has global scope, and a local variable `var` declared within the function. However, global `var` and local `var` are completely independent variables; they don't have to be the same type, and they won't interfere with each other. Likewise, the global variable a and the local variable a share only a name; you can modify the local a as much as you like, without modifying the global a.

It helps to know how names are looked up in C++. First the local scope is examined and then any *enclosing* local scopes. Only then does C++ examine global scope. If you really need to access the hidden variable `var`, you can use the *global scope operator* (`::`), as in the following example:

```
;> void another_test() { int var = 0; cout << ::var << endl; }
;> another_test();
1
```

Recall that uninitialized variables are not guaranteed to have a 'sensible' value. Global variables will initially be zero, but local variables initially have arbitrary values. This is a common source of problems, and it causes bugs that mysteriously come and go. As much as possible, you should initialize local variables and generally keep the declaration as close to the first use of the variable as possible. This example shows a typical symptom of an uninitialized local variable:

```
void show_var() {
 int ii;
 cout << ii << endl;
}
;> show_var();
7521584
```

It is important to realize that local variables are by default *volatile*, which means they do not keep their values after the function has finished executing. (Actually, they may randomly keep their values, which is even worse than not keeping their values.) This may appear to be a strange default (and some languages don't do it), but it is a consequence of how C++ allocates local variables. Later you will see why this is the case, (in the section "Recursion" in Chapter 6), but for now, don't depend on local variables to remember anything.

Note that a variable might not be in scope but still be alive and well. For instance, the following example depends on the second function not forgetting the value of `ii` while the first function is called:

```
void second(int I) { cout << I << endl; }
void first() {
  Int ii = 2;
  Second(ii);
  Cout << ii << endl;
}
```

If you want a local variable that is not volatile, you can use the `static` qualifier with a variable declaration. This kind of variable is initialized once, at the start of the program, and it keeps its value until the next call of the function. In this example, the function `show_it()` has a local variable `ii`. But because it has been declared as `static`, this variable will remember its value:

EXAMPLE

```
void show_it() {
 static int ii = 0;
 ii++;
 cout << ii << endl;
 }
;> show_it();
1
;> show_it();
2
;> show_it();
3
```

Case Study: A Bug and Defect Tracking System

The case studies that appear at the ends of most chapters of this book present nontrivial examples of the C++ language in use. You will get more experience at reading C++ and how to actually use it in practice. Along the way, you will learn how to go about the business of writing software.

The Specification

Bugs are a fact of life in any software project, no matter how clean you kept the surfaces and how shut the windows were, but the testing phase is meant to find them. It is a good idea to keep track of bugs. Traditionally, programmers write themselves notes about bugs on small pieces of paper, but this method breaks down very quickly.

In this section you will write a very straightforward system for keeping track of bugs. The following information is crucial: when the bug was discovered, how severe it is, and a short description. Most bugs are fixed eventually, so the system must be able to retire bug reports; however, you need to keep a record of bugs fixed so that the programmers can prove that they have been productive. The system needs to report all bugs with a severity or level greater than a specified value. Finally, the system must report all bugs within a specified date range. Armed with this specification of the system, you can begin to start work on the system.

Top-Down Design

In programming, as in most things with computers, it pays to think before you type. There are many ways to start designing programs, but with any approach, it is important not to immediately start writing code (unless it really is a trivial program or you have done something very similar that could be reused). One way to plan a system is to break the system into major functions. There appear to be three of these for the bug tracking program you need to develop: `add_record()`, `copy_and_remove()`, and `show_and_filter()`.

Bottom-up Coding: Manipulating Dates

While I am thinking about the top-level design of a system, I find it useful to start writing code to do lower-level tasks. This keeps me from prematurely writing the program, and it gives me an idea of what is needed to actually implement the actions. In the case of the bug tracking application, you might start by thinking about dates; you need to be able to show and compare calendar dates. One way of doing this is to keep the dates as strings in some standard format. There is some dispute across the Atlantic about whether the month or the day goes first, so I'll please neither and pick the International Standards Organization standard date format, which shows June 28, 2001, as 2001-06-28. I have supplied a function, `str2int()`, which converts a number as a string into its value. Using `str2int()`, you can immediately write functions that extract the year, month, and date from an ISO format date:

EXAMPLE

```
// don't worry about str2int() yet
int str2int(const string& s) { return atoi(s.c_str()); }

// All date routines assume ISO format: YYYY-MM-DD
int year(string s)
{
  return str2int(s.substr(0,4));
}
int month(string s)
{
  return str2int(s.substr(5,2));
}

int day(string s)
{
  return str2int(s.substr(8,2));
}
```

After you break up a date in this way, it's straightforward to compare two dates. The strategy is as follows, given two dates: First, compare the year part. If it's different, you know the dates are different and don't need to continue comparing. Next, compare the month part and then compare the day part. The following routine `compare_dates()` returns zero if two dates are the same, negative if the first date is less than the other, and positive if the first date is more than the other. Notice how the `return` statement allows you to leave the function as soon as you have a definite answer:

```
int compare_dates(string d1, string d2)
{
 int yd = year(d1) - year(d2);
 if (yd != 0) return yd;
 int md = month(d1) - month(d2);
 if (md != 0) return md;
 int dd = day(d1) - day(d2);
 if (dd != 0) return dd;
 return 0;  // we are equal
}
```

Just as a bit of fun, consider the problem of showing `2001-06-28` as `28 Jun 2001`. You have to use the month index (1 to 12) to return the three-letter abbreviation. Obviously, a `switch` statement would do the job, but the following is a little more elegant:

```
const string _MONTHS_ =
 "JanFebMarAprMayJunJulAugSepOctNovDec";
string month_as_str(int idx)
{
```

```
// idx is 1-12
  return _MONTHS_.substr(3*(idx-1),3);
}
```

Appending a Bug Report

For this version of the program, we will use plain ASCII text files for the bug reports; text files have advantages as well as disadvantages. Their chief advantage is that they are easy to read by humans. (At the end of Chapter 5, I'll show you a version of this application that uses binary files.) An output file can be opened for appending by using the `ios::app` constant in the `open()` method of `ofstream`. The following function thus adds another line to the file of bug reports:

```
void add_record(string file, int id, int level,
                string date, string description)
{
  ofstream out;
  out.open(file.c_str(),ios::app);
  out << id << ' ' << level << ' ' << date
      << ' ' << description << endl;
  out.close();
}
```

Each bug report is given an identifying (and unique) number. This is not part of the specification, but it's usually a good idea to number things, and it is easy to refer to a particular bug this way. But how do you apply a unique number to each bug? Rather than rely on the user to remember the last ID, the following routine reads the ID from a file, increments it, and writes it back:

EXAMPLE

```
const string ID_FILE = "_ID-FILE_.TXT";
int next_id()
{
  int id;
 //...read the last id from the id file
  ifstream in;
  if (in.open(ID_FILE.c_str())) { // it does exist..
    in >> id;
    in.close();
  } else id = 0;
  id++;
 //...write the new value back...
  ofstream out;
  out.open(ID_FILE.c_str());
  out << id;
  out.close();
  return id;
}
```

Discarding Bug Reports

Eventually, a large project generates hundreds of bug reports, and it is necessary to remove the reports when the bugs are fixed. You could keep a record of dead bugs by diligently creating backups of the bugs file, but this is not really convenient for analysis. Instead, you can copy all bugs marked as dead to a discards file and then copy the rest of the bugs to a temporary file. When you are done copying, you can either copy the temporary file back or simply rename it. In the following example, you use a file copy routine:

```
void copy_file(string in_file, string out_file)
{
 ifstream in;
 ofstream out;
 string line;

 in.open(in_file.c_str());
 out.open(out_file.c_str());

 while (! in.eof()) {
   getline(in,line);
   out << line << endl;
 }

 in.close();
 out.close();
}
```

As long as you are not at the end of the input file, you continue to read a line at a time and write it to output. Having defined this operation, you can now write the function that discards bugs:

```
const string TMP_FILE = "TMP-BUG-FILE.TXT";
void copy_and_remove(string bug_file,string discard_file,
                   string id_list)
{
  ifstream bugs_in;
  ofstream discards;
  ofstream bugs_out;
  istringstream ids(id_list);
  int id, target_id;
  string line, id_str;

  bugs_in.open(bug_file.c_str());
  bugs_out.open(TMP_FILE.c_str());
  discards.open(discard_file.c_str());
```

```
 // we have to get the first target id (at least one)
  ids >> target_id;

  while (bugs_in >> id) {
  //.....not interested in rest of line...
     getline(bugs_in,line);
  //...Dump to discards and fetch our next target..
    if (id == target_id) {
       discards << id << line << endl;
       if (! ids.eof()) ids >> target_id;
    }
    else  bugs_out << id << line << endl;
  }

  bugs_in.close();
  bugs_out.close();
  discards.close();
  ids.close();

  // And finally, overwrite the original with the temporary file
  copy_file(TMP_FILE, bug_file);
}
```

The idea here is that the function is passed a string that contains a list of IDs. You can get these IDs in more than one way, but a particularly convenient method is to use an `istringstream` object to read them one by one. Before you start reading the file, the first ID is read from the string, and it becomes the first target. Each ID is read in, followed by the rest of the line (you don't need the extra fields, and it would be inefficient to read them in individually). If the ID matches the target, that bug report must go to the discards file, and you read the next target from the string. It is important to check the result of reading in the bug ID, so that you know when the file has been fully processed.

Showing the Report

Users want to place two kinds of constraints on bug reports: level and date range. A report can have hundreds of bugs, so it's important that the report pause every 20 lines or so to ask the user whether he or she wants to see more. This is a good example of using a `bool` as a *logical flag*. The flags `level_filter` and `date_filter` are true only if the level and date inputs are nontrivial; in our application, the levels begin at one, so zero means no level. If any of these flags is true, then the corresponding test is performed, as follows:

```
void show_and_filter(string bfile, int min_level,
                     string lower_date, string upper_date)
{
   ifstream in;
   int id,level,lcount=1;
   string date,description;
   bool level_filter = min_level > 0;
   bool date_filter = lower_date != "";
   bool pass;

   in.open(bfile.c_str());
   while (in >> id >> level >> date) {
      in.getline(description);

//  The idea here is that these conditions only kick in
    //  if we did want to filter on level and/or date...
      pass = true;
      if (level_filter) pass = pass && level >= min_level;
      if (date_filter) pass = pass &&
         compare_dates(lower_date,date) < 0 && compare_dates(upper_date,date) > 0;
      if (pass) {
           cout << id << ' ' << level << ' ' << date << ' '
                << description << endl;
           lcount++;
      }

     //  Output 20 lines at a time....
      if (lcount % 20 == 0) {
        string tmp;
        cout << "Press return to continue....\n";
        cin.getline(tmp);
      }
   }
   cout << "There were " << lcount << " records found\n";
   in.close();
}
```

Putting It All Together

Each one of the operations in this case study's application can be run from the UnderC prompt, and that's how we test the system: Each part is exercised individually. For example, by the time you used `compare_dates()`, you had tested it on a few cases. But, you can't expect users to type function names, and most C++ systems don't support that style of interaction. A basic user interface involves a simple menu of choices that is printed out before each user command. You can then use a switch statement to handle the commands. Note that the `switch` statement is within a `for(;;)` loop;

this is the traditional way of writing a loop that continues forever. Eventually the user will get bored and enter **5**, and this causes the loop to directly return from the main function. A real system would have a lot more error checking than the example here; for example, it would check whether any entered dates are valid. The `show_menu()` function prints out the menu lines (note the use of `'\n'` rather than `endl`) and allows the user to enter a number, which is then returned. The chief function of this program is `do_bugs()`, which is basically a `switch` statement that does a `return` when the user enters the number 5. Until then, it will keep calling `show_menu()`. Each `case` label is followed by a block; I've done this so that the code for each option can have its own local variables.

EXAMPLE

```
int show_menu()
{
  cout << "1. Add a bug report\n"
       << "2. Show bugs with specified level\n"
       << "3. Show bugs within a time period\n"
       << "4. Delete list of bugs\n";
       << "5. Quit\n";
  int val;
  cin >> val;
  return val;
}

void do_bugs()
{
  for(;;) { // loop 'forever'
    int icmd = show_menu();
    switch(icmd) {
    case 1: {
      string date,descript;
      int id, level;
      cout << "give: level date" << endl;
      cin >> level >> date;
      cout << "give:  description" << endl;
      getline(cin,descript);
      if (date == "*") date = today;
      id = next_id();
      add_record(BF,id,level,date,descript);
      cout << "bug " << id << " added...\n";
    } break;
    case 2: {
      int level;
      cout << "give minimum level\n";
      cin >> level;
      show_and_filter(BF,level,"","");
    } break;
    case 3: {
      string d1,d2;
      cout << "give lower and upper dates\n";
      cin >> d1 >> d2;
      show_and_filter(BF,0,d1,d2);
    } break;
    case 4: {
      string ids;
      cout << "give IDs in ascending order\n";
      getline(cin,ids);
      copy_and_remove(BF,DF,ids);
    } break;
    case 5: return;
    default: cout << "unrecognized command\n";
    }
  }
}
```

What's Next

This chapter has shown you C++ as a programming language; you have collected statements as functions and learned how to conditionally do statements or construct loops using control statements such as `if-else`, `while`, `do-while`, `for`, and `switch`. The next chapter will deal how to organize data using C++ arrays and standard containers. Routine calculations like finding the maximum value of a set of numbers can be easily done using the standard algorithms.

Arrays and Algorithms

So far we have discussed the core arithmetic and logical operations of C++, as well as features such as strings and input /output that are part of the standard library. In the case study in Chapter 2, “Functions and Control Statements,” we hit some limitations of basic C++. To pass a number of integers as a parameter, we were forced to write them as a string. There are two problems with this; first, there is no guarantee that these are all valid integers, and second, this would be very inefficient for really large lists of integers. That case study also depends heavily on having everything like ID lists in the correct order, but we cannot control the world, and often things happen in an order other than what we plan, so we cannot insist on sorted lists. C++ provides a basic built-in way to set up tables of numbers or objects, called *arrays*. The standard library also provides more sophisticated ways to group items together, called the *standard containers*.

Algorithms are standard procedures (or recipes) for doing operations such as searching and sorting. I will compare two ways to search tables of values (binary and linear search), and I will introduce the very powerful library of general algorithms in the standard C++ library.

In this chapter you will learn

- How to declare, initialize, and use arrays
- How to search for particular values and how to sort arrays
- How to use the standard containers, including vectors, lists, and maps

Arrays

Arrays contain a linear table of values, which must all be of the same type, called the *base type*. These values are called the array's *elements*, and any element can be specified by its position in the array, called its *index*. In this section we will discuss C++'s built-in arrays, together with typical array operations like copying and searching. The next section will then look at the standard containers, which are 'intelligent' arrays.

Arrays as Tables of Values

A C++ array declaration reserves space for a number of elements. For instance, the following declarations create an array of `int`s and an array of `char`s:

```
;> int arr1 [10];
;> char arr2 [10];
;> sizeof(arr1);
(int) 40
;> sizeof(arr2);
(int) 10
```

You can see in this example that in general, the space occupied by an array is the number of elements multiplied by the size of each element. The number of elements in an array is usually called its *dimension*, and it can be any positive constant. You can declare more than one array at a time; in the following code, notice that the dimension follows each new array name. The dimension of `a3` is the expression `M*N`, which is acceptable because it contains no variables. You can mix regular variables in as well, although this is not considered a stylish thing to do. The variable `x` as declared here is just a plain `double`.

```
;> const int N = 20, M = 30;
;> double a1[6],a2[N],a3[M*N],x;
```

You declare two-dimensional arrays (which are used for representing tables or mathematical matrixes) with two dimensions, rather than with a comma-separated pair, as in the following example:

```
;> int table[10][10];  // NOT int table[10,10]!!
```

Initializing Arrays

You can initialize arrays by using a list of values separated by commas enclosed in braces, as in the following declaration of `arr`. The declaration of `names` shows how you can optionally leave out the constant in the brackets and let the system work out the dimension from the size of the list. For global declarations, the values in the list must be constants.

```
;> int arr[4] = {10,5,6,1};
;> string names[] = {"Peter","Alice","James"};
;> double interest_rates[]
  = {
     5.1,
     5.6,
     6.0,
     7.0
    };
;> int numbers[] = {1,2,3,4,5,6,7,8};
```

You cannot assign arrays in the following fashion because C++ arrays are not variables. You also cannot use `a1 = a2`, where both `a1` and `a2` are arrays.

```
;> arr = {1,2,3,5};
CON 10:parse error
CON 10:error in expression
```

You can, however, treat each element of the array as if it is a variable; this is called *indexing* the array, and the index is often also called a *subscript*. To access the i-th element of `arr` you say `arr[i]`, where the index `i` is put in square brackets (`[]`) like in other languages such as Pascal. This expression can be used wherever a variable can be used. The array index goes from 0 to one less than the size of the array. Using the definitions of `arr` and `names` from the preceding examples, you can say the following:

EXAMPLE

```
;> arr[0];
(int&) 10
;> arr[3];
(int&) 1
;> arr[0] + 2*arr[3];
(int) 12
;> arr[2] = 2;
(int&) 2
;> names[2] + " Brown";
(string) 'James Brown'
```

You can make constant arrays that cannot be written to afterward:

```
;> const int AC[] = {10,2,4};
;> AC[2] = 2;
CON 71:Can't assign to a const type
```

There are two basic rules to keep in mind with C++ arrays: Arrays go from 0 to `n-1` (not 1 to `n`), and there is no bounds checking whatsoever with arrays. Therefore, if you write past the end of an array, the value will probably go somewhere else. In the following case, we have changed the value of `var`, (which was an innocent bystander) by forgetting both rules:

```
;> int a1[2];
;> int var;
;> a1[2] = 5;
(int&) 5
;> var;
(int) var = 5
```

NOTE

The fact that C++ arrays always begin at 0 is a benefit. In some other languages, you can change the start value by a global setting.

NOTE

Running over the end of an array is particularly bad if the array is declared local to a function, and on most systems, this will crash the program badly. The first C program I ever wrote did this, and my DOS machine went down in flames. Up to then I had trained on Pascal, which slaps you on the wrist if you exceed array bounds, but it doesn't amputate the whole hand. You may regard this as dumb and dangerous behavior, and many would agree with you, but C was designed to be fast and somewhat reckless.

You initialize most arrays by using a `for` loop. `for` loops naturally go from 0 to `n-1`, as required. This code is equivalent to the initialization of the array `numbers` from an earlier example in this chapter:

```
;> for(int i = 0; i < 8; i++) numbers[i] = i+1;
```

Arrays make possible code that otherwise would require big ugly `switch` statements. Consider the problem of starting with a date in International Standards Organization (ISO) form, and working out the day number (counting from January 1). One solution begins like this and would require quite a bit more tedious coding:

```
   int days = days(date);
   switch(month(date)) { // runs from 1 to 12
   case 2: days += 31;  break;
   case 3: days += 61;  break;
    ....
   }
```

Another solution would be to initialize an array with the lengths of the months and then generate a running sum. Thereafter, you could use just one line. Here's how you would implement this solution:

EXAMPLE

```
;> int days_per_month[] = {0,31,28,31,30,31,31,30,31,30,31};
;> int days_upto_month[13];
;> int sum = 0;
;  int days_upto_month[0] = 0;
;> for(int i = 1; i <= 12; i++)
  days_upto_month[i] = days_upto_month[i-1] + days_per_month[i];
;> int days = days_upto_month[month(day)] + day;
```

Passing Arrays to Functions

You can pass an array to a function, by declaring an array argument with an empty dimension. You usually have to pass the size as another argument because there is no size information in C/C++ arrays. The advantage of this is that the following function can be passed any array of `int`s:

```
void show_arr(int arr[], int n)
{
  for(int i = 0; i < n; i++) cout << arr[i] << ' ';
  cout << endl;
}
;> show_arr(numbers,8);
1 2 3 4 5 6 7 8
```

However, you cannot pass this function an array of `double`s. The following error message means that the compiler failed to match the actual argument (`double[]`) with the formal argument (which was `int[]`):

```
;> double dd[] = {1.2, 3.4, 1.2};
;> show_arr(dd,3);
CON 28:Could not match void show_arr(double[],const int);
```

A `double` value is organized completely differently from an `int` type. If the compiler allowed you to pass an array of `double`s, you would get very odd integers displayed. On the other hand, simply passing a `double` value is not much trouble in C++; the compiler sensibly converts the `double` to an integer and gives you a mild warning that this will involve a loss of precision. This difference between passing a `double` value and passing an array of `double`s exists because in C++, ordinary variables (that is, *scalars*) are passed *by value*; they are actually copied into the formal argument. Arrays, on the other hand, are passed *by reference*. This difference is similar to the difference between mailing someone his or her own copy of a document and telling the person where to find it on a network. An interesting result of being able to pass arrays by reference is that you can change an array's elements within a function. (If you altered the document on the network, this would change that document for all readers.) The following small function modifies the first element of its array argument:

```
;> void modify_array(int arr[])  { arr[0] = 777; }
;> modify_array(numbers);
;> show_arr(numbers,8);
777 2 3 4 5 6 7 8
```

What if you didn't want the array to be (accidently) modified? This was always an issue with C, but C++ has a number of solutions. One solution is to use the standard library's vector class, which we will discuss later in this chapter. Another good solution is to make the array parameter `const`, as in the following example, so that it is impossible to modify the parameter within the function:

```
void just_looking(const int arr[]) {
  cout << arr[0] << endl;
}
```

A `const` parameter is a promise from the function to the rest of the program: "Pass me data, and I shall not modify it in any form." It's as if I tell you where to find the document on the server, but make it read-only.

Reading Arrays

Data arrays are commonly read from files. Because arrays are passed by reference, it is easy to write a function to do this job. The following example uses a few new shortcuts:

EXAMPLE

```
int read_arr(string file, int arr[], int max_n)
{
  ifstream in(file.c_str());
  if (in.bad()) return 0;
  int i = 0;
  while (in >> arr[i]) {
      ++i;
      if (i == max_n) return i;
  }
  return i;
}
```

In this example, if you initialize the `ifstream` object `in` with a filename, that file will be automatically opened. The file is automatically closed when `in` goes out of scope. Also, the `bad()` method tells you if the file cannot be opened. There is no simple way to work out how many numbers are in any given file, so there's always the danger that you might overrun an array. In functions that modify arrays, it is common for the actual allocated array size to be passed. This function reads up to `max_n` numbers and will exit the file read loop if there are more numbers; it will return the number of values read, which is one plus the last index.

Writing out arrays to a file is straightforward. You can use the remainder operator (`%`) to control the number of values written out per line, as in the following example. `(i+1) % 6` is zero for i = 5,11,17,...—that is, the condition is true for every sixth value. (I added 1 to `i` so that it would not put out a new line for `i == 0`.)

EXAMPLE

```
void write_arr(string file, int arr[], int n)
{
   ofstream out(file.c_str());
   for(int i = 0; i < n; i++) {
     out << arr[i];
     if ((i+1) % 6 == 0) out << endl;
                    else out << ' ';
   }
   out << endl;
}
```

Searching

You will find that you often need to search for a value in a table of numbers. You want to know the index of that value within that table, or even simply whether the value is present. The simplest method is to run along until you find the value, or *key*. If you don't find the key, you just return `-1` to indicate that the key is not present in the array, as in the following example, where the function `linear_search()` is defined:

```
int linear_search(int arr[], int n, int val)
{
 for(int i = 0; i < n; i++)
   if (arr[i] == val) return i;
 return -1;
}
;> linear_search(numbers,4);
(int) 3
;> linear_search(numbers,42);
(int) -1
```

This method is fast enough for small tables, but it isn't adequate for large tables that need to be processed quickly. A much better method is to use a *binary search*, but the elements of that table must be in ascending order. To perform a binary search, you first divide the table in two; the key must be either in the first half or the second half. You compare the key to the value in the middle; if the key is less than the middle value, you choose the first half, and if the key is greater than the middle value, you choose the second half. Then you repeat the process, dividing the chosen half into halves and comparing the key, until either the key is equal to the value or until you cannot divide the table any further. Here is a C++ example of performing a binary search:

```
int bin_search(int arr[], int n, int val)
{
  int low = 0, high = n-1;    // initially pick the whole range
  while (low <= high) {
    int mid = (low+high)/2;  // average value...
    if (val == arr[mid]) return mid;   // found the key
    if (val < arr[mid]) high = mid-1;  // pick the first half
                 else   low  = mid+1;  // pick the second half
  }
  return -1;  // did not find the key
}
```

How much faster is a binary search than a linear search? Well, if there are 1,000 entries in a table, a binary search will take only about 30 tries (on average) to find the key. The catch here is that the table must be sorted. If it's already sorted, then it seems a good idea to insert any new value in its proper place. Finding the place is easy: You run along and stop when the key is less than the value. The code for finding the position is similar to the code for a linear search, but `-1` now means that the key was larger than all the existing values and can simply be appended to the table:

```
int linear_pos(int arr[], int n, int val)
{
 for(int i = 0; i < n; i++)
   if (arr[i] > val) return i;
 return -1;
}
```

Inserting

It takes a surprising amount of effort to insert a new value into an array. To make space for the new value means moving the rest of the array along. For example, consider that inserting 4 into the sequence `1 3 9 11 15`; `linear_pos()` gives an index of `2` (because 9 is greater than 4). Everything above that position must be shifted up to make room; the top line shows the array subscripts:

0	1	2	3	4	5	subscripts
1	3|	9	11	15		before shifting up by one
1	3	xxx	9	11	15	after shifting up by one

Here I've shown the sequence before and after the shift. We have to put 4 before index 2 (that is, the third position). So everything from the third position up (`9 11 15`) has got to move up one. That is, the shift involves `A[5] = A[4]`, `A[4] = A[3]`, and `A[3] = A[2]`. We can then set `A[2] = 4`. In general:

EXAMPLE

```
void insert_at(int arr[], int n, int idx, int val)
{
  if (idx==-1) arr[n] = val;  // append
  else {
    for(int i = n; i > idx; i—)
       arr[i] = arr[i-1];
    arr[idx] = val;
  }
}
;> int n = 5;
;> int pos = linear_pos(arr,n,4);
;> insert_at(arr,n,pos,4);
;> show_arr(arr,n);
1 3 4 9 11 15
```

Once `insert_at()` is defined, then it is straightforward to put a new element into an array so that it remains in order. The special case when `idx` is –1 just involves putting the value at the end. But generally there will be a lot of shuffling, making insertion much slower than array access. Note that this routine breaks the first rule for dealing with arrays: It goes beyond the end of the array and writes to `arr[n]`. For this to work, the array needs to have a dimension of at least `n+1`.

This is a good example of a `for` loop doing something different from what you've seen `for` loops do before: in this case, the `for` loop is going down from `n` to `idx+1`. In tricky cases like this, I encourage you to experiment (that's what UnderC is for.) But nothing beats sitting down with a piece of paper and a pencil.

As you can see from the examples in this section, it isn't straightforward to insert a value into an array in order. The algorithm for deleting an item is similar to `insert_at()` and involves shifting to the left; I leave this as an exercise for a rainy afternoon.

Sorting

Sorting a sequence that is not in any order also involves moving a lot of data around. The following algorithm is called a *bubble sort* because the largest numbers "bubble" up to the top. It includes calls to `show_arr()` so that you can see how the larger numbers move up and the smaller numbers move down:

EXAMPLE

```
void bsort(int arr[], int n)
{
  int i,j;
  for(i = 0; i < n; i++)
    for(j = i+1; j < n; j++)
      if (arr[i] > arr[j]) { // swap arr[i] and arr[j]
        show_arr(arr,n);  // print out array before swap
        int tmp = arr[i];
        arr[i] = arr[j];
        arr[j] = tmp;
      }
  show_arr(arr,n); // show sorted array
}
```

```
;> int b[] = {55,10,2,3,6};
;> bsort(b,5);
55 10 2 3 6
10 55 2 3 6
2 55 10 3 6
2 10 55 3 6
2 3 55 10 6
2 3 10 55 6
2 3 6 55 10
2 3 6 10 55
```

Containers

Arrays have several disadvantages. Earlier in this chapter we discussed their lack of size information, which means you must use two arguments to pass an array to a function. It also means that you cannot check an array index at runtime to see whether it's out of bounds. It is easy to crash a program by using the wrong index; what is perhaps worse—because the program seems to work—is that memory can be silently overwritten. All C programmers will tell you that these are some of the worst bugs to solve. Built-in arrays are also inflexible in that they have a fixed size that must be a constant. Although it is very fast to access array data randomly, insertions and deletions are slow.

The standard library defines a number of container types. A *container* holds a number of elements, like an array, but it is more intelligent. In particular, it has size information and is resizable. We will discuss three kinds of standard containers in the following sections: `vector`, which is used like a built-in array, but is resizeable; `list`, which is easy to insert elements into; and `map`, which is an *associative array*. That is, it associates values of one type with another type.

Resizable Arrays: `std::vector`

You use the `vector` container type the same way you use an ordinary array, but a `vector` can grow when required. The following is a `vector` of 10 `int`s:

```
;> vector<int> vi(10);
;> for(int i = 0; i < 10; i++) vi[i] = i+1;
;> vi[5];
(int&) 6
;> vector<int> v2;
;> v2.size();
(int) 0
;> v2 = vi;
;> v2.size();
(int) 10
```

vector is called a *parameterized type*. The type in angle brackets (<>) that must follow the name is called the *type parameter*. vector is called parameterized because each specific type (vector<int>, vector<double>, vector<string>, and so on) is built on a specific base type, like a built-in array. In Chapter 10, "Templates," I will show you how you can build your own parameterized types, but for now it's only important that you know how to use them.

vi is a perfectly ordinary object that behaves like an array. That is, you can access any element very quickly using an index; this is called *random access*. Please note that the initial size (what we would call the array dimension) is in parentheses, **not** in square brackets. If there is no size (as with v2) then the vector is initially of size zero. It keeps its own size information, which you can access by using the size() method. You cannot initialize a vector in the same way as an array (with a list of numbers), but you can assign them to each other. The statement v2 = vi actually causes all the elements of vi to be copied into v2. A vector variable behaves just like an ordinary variable, in fact. You can pass the vi vector as an argument to a function, and you won't need to pass the size, as in the following example:

```
void show_vect(vector<int> v)
{
  for(int i = 0; i < v.size(); i++) cout << v[i] << ' ';
  cout << endl;
}
;> show_vect(vi);
1 2 3 4 5 6 7 8 9 10
```

You can resize the vector vi at any point. In the following example the elements of vi are initialized to random numbers between 0 and 99. (n % 100 will always be in that range). vi is then resized to 15 elements:

EXAMPLE

```
;> for(int i = 0; i < 10; i++) vi[i] = rand() % 100;
;> show_vect(vi);
41 67 34 0 69 24 78 58 62 64
;> vi.resize(15);
show_vect(vi);
41 67 34 0 69 24 78 58 62 64 0 0 0 0 0
```

You can resize the vi vector without destroying its values, but this can sometimes be quite a costly operation because the old values must be copied. Note that vectors are passed to functions by value, not by reference. Remember that passing by value involves making a copy of the whole object. In the following example, the function try_change() tries to modify its argument, but doesn't succeed. Earlier in this chapter ("Passing Arrays to Functions") you saw a similar example with built-in arrays, which did modify the first element of its array argument.

EXAMPLE

```
;> vector<int> v2 = vi;
;> v2.size();
(int) 15
;> v2[0];
(int&) 41
;> void try_change(vector<int> v) { v[0] = 747; }
;> try_change(v2);
;> v2[0];
(int&) 41
```

At this point, you may be tired of typing `vector<int>`. Fortunately, C++ provides a shortcut. You can create an alias for a type by using the `typedef` statement. The form of the `typedef` statement is just like the form of a declaration, except the declared names are not variables but type aliases. You can use these `typedef` names wherever you would have used the original type. Here are some examples of how to use `typedef`, showing how the resulting aliases can be used instead of the full type:

EXAMPLE

```
;> typedef unsigned int uint;
;> typedef unsigned char uchar;
;> typedef vector<int> VI;
;> typedef vector<double> DV;
;> uint arr[10];
;> DV d1(10),d2(10);
;> VI v1,v2;
;> int get(VI v, int i) { return v[i]; }
```

Think of `typedef` names as the equivalent of constants. Symbolic constants make typing easier (typing pi to 12 decimal places each time is tedious) and make later changes easier because there is only one statement to be changed. In the same way, if I consistently use `VI` throughout a large program, then the code becomes easier to type (and to read). If I later decide to use some other type instead of `vector<int>`, then that changes becomes straightforward.

As you have learned, passing a `vector` (or any standard container) to a function involves a full copy of that `vector`. This can make a noticeable difference to a program's performance if the function is called enough times. You can mark an argument so that it is passed by reference, by using the address operator (`&`). You can further insist that it remains constant, as we did earlier in the chapter for arrays and as shown in the following example:

```
void by_reference (vector<int>& vi)
 { vi[0] = 0; }
void no_modify (const vector<int>& vi)
 { cout << vi[0] << endl; }
```

Generally, you should pass `vectors` and other containers by reference; if you need to make a copy, it's best to do it explicitly in the function and make such reference arguments `const`, unless you are going to modify the vector. When experienced programmers see something passed by reference, they assume that someone is going to try to change it. So the preferred way of passing containers is by `const` reference, as in the preceding example. You can always use the `typedef` names to make things look easier on the eye, as shown here:

```
int passing_a_vector (const VI& vi) { return vi[0]; }
```

The standard string is very much like a `vector<char>`, and it is considered an "almost container." Strings can also be indexed like `arrays`, so if `s` is a string, then `s[0]` would be the first character (**not** substring), and `s[s.size()-1]` would be the last character.

Linked Lists: `std::list`

`vectors` have strengths and weaknesses. As you have seen, any insertion requires moving elements, so if a `vector` contained several million elements (and why not?), insertion could be unacceptably slow. Although `vectors` grow automatically, that process can also be slow because it involves copying all the elements in the `vector`.

Lists are also sequences of elements, but they are not accessed randomly, and they are therefore not like arrays. Starting with an empty list, you append values by using `push_back()`, and you insert values at the front of the list by using `push_front()`. `back()` and `front()` give the current values at each end. To remove values from the ends, you use `pop_front()` and `pop_back()`. The following is an example of creating a list:

EXAMPLE

```
;> list<int> li;
;> li.push_back(10);
;> li.push_front(20);
;> li.back();
(int) 10
;> li.front();
(int) 20
;> li.size();
(int) 2
;> li.pop_back();
;> li.back();
(int) 20
```

You can remove from a list all items with a certain value. After the remove operation, the list contains only `"two"`:

```
;> list<string> ls;
;> ls.push_back("one"); ls.push_back("two"); ls.push_back("one");
;> ls.remove("one");
```

Associative Arrays: `std::map`

In mathematics, a *map* takes members of some input set (say `0..n-1`) to another set of values; a simple example would be an array. The standard C++ `map` is not restricted to *contiguous* (that is, consecutive) values like an array or a vector, however. Here is a simple `map` from `int` to `int`:

```
;> map<int,int> mii;
;> mii[4] = 2;
(int&) 2;
;> mii[88] = 7;
(int&) 7
;> mii.size();
(int) 2
;> mii[4];
(int&) 2
;> mii[2];
(int&) 0
```

You access `map`s the same way you access arrays, but the key values used in the subscripting don't have to cover the full range. To create the `map` in the preceding example by using arrays, you would need at least 89 elements in the array, whereas the `map` needs only 2. If you consider a `map` of phone numbers and contact names, you can see that an ordinary array is not an option. `map`s become very interesting when the key values are non-integers; we say that they associate strings with values, and hence they are often called *associative arrays*. Typically, a `map` is about as fast as a binary search.

```
;> map<int,string> mis;
;> mis[6554321] = "James";
(string&) "James";
;> mis.size();
(int) 1
;> map<string,int> msi;
;> msi["James"] = 6554321;
(int&) 6554321
;> msi.size();
(int) 1
;> msi["Jane"];
(int&) 0
;> msi.size();
(int) 2
```

Something that is important to note about maps is that they get bigger if you are continuously querying them with different keys. Say you are reading in a large body of text, looking for a few words. If you are using array notation, each time you look up a value in the map, the map gets another entry. So a map of a few entries can end up with thousands of entries, most of which are trivial. Fortunately, there is a straightforward way around this: You can use the map's `find()` method. First, you can define some `typedef` names to simplify things:

```
;> typedef map<string,int> MSI;
;> typedef MSI::iterator IMSI;
;> IMSI ii = msi.find("Fred");
;> ii == msi.end();
(bool) true
```

The `find()` method returns a map iterator, which either refers to an existing item or is equal to the end of the map.

Maps are some of the most entertaining goodies in the standard library. They are useful tools, and you can use them to write very powerful routines in just a few lines. Here is a function that counts word frequencies in a large body of text (testing this case, the first chapter of Conan Doyle's *Hound of the Baskervilles*, courtesy of the Gutenberg Project):

EXAMPLE

```
int word_freq(string file, MSI& msi) {
   ifstream in(file.c_str());
   string word;
   while (in >> word) msi[word]++;
   return msi.size();
}
;> word_freq("chap1.txt",msi);
(int) 945
;> msi["the"];
(int&) 94
```

This example uses the shorthand for opening a file, and it assumes that the file will always exist. The real fun happens on the fourth line in this example. For each word in the file, you increment the map's value. If a word is not originally present in the map, `msi[word]` is zero, and a new entry is created. Otherwise, the existing value is incremented. Eventually, `msi` will contain all unique words, along with the number of times they have been used. This example is the first bit of code in this book that really exercises a machine. The UnderC implementation is too slow for analyzing large amounts of text, but Chapter 4, "Programs and Libraries," shows how to set up a C++ program that can be compiled into an executable program.

Stacks and Queues

Sometimes it's useful to build up a `vector` element by element. This works exactly like adding to the end of a `list`. You can add new elements at the end with `push_back()`; `back()` gives the value of the last value of the vector, and `pop_back()` removes the last value, decrementing the size.

EXAMPLE

```
;> typedef vector<int> VI;
;> VI vs;
;> vs.push_back(10);
;> vs.push_back(20);
;> show_vect(vs);
10 20
;> vs.size();
(int) 2
;> vs.back();
(int) 20
;> vs.pop_back();
;> vs.back();
(int) 10
;> vs.size();
(int) 1

void read_some_numbers(VI& vi, string file) {
  int val;
  ifstream in(file.c_str());
  while (in >> val)  vi.push_back(val);
}
```

Often you are given input without any idea of how many numbers to expect. If you use `push_back()`,the vector automatically increases in size to accommodate the new numbers. So the function `read_some_numbers()` will read an arbitrary number of integers and add them to the end of the `vector`.

There is no `push_front()` method because that would potentially be an expensive operation. If you really need to do it, you can use `vi.insert(vi.begin(),val)`.

The operations `push` and `pop` define a *stack*. A stack is similar to the spring-loaded device often used for dispensing plates in cafeterias. As you remove plates from the top of the device (that is, "pop the stack"), more plates rise and are ready to be taken. You can also push plates onto the pile. A stack operates in first-in, last-out (FILO) fashion: if you push 10, 20, and 30, then you will pop 30, 20, and 10. Stacks are one of the basic workhorses of computer science, and you see them all over the place. A common use is to save a value, as in the following example:

EXAMPLE

```
void push(int val) {
  vi.push_back(val);
}
int  pop() {
  int val = vi.back();
  vi.pop_back();
  return val;
}
;> int val = 1;
;> push(val);     // save val;
;> val = 42;      // modify val;
(int) 42
.... do things with val.......
;> val = pop();   // restore val;
```

A *queue*, on the other hand, operates in first-in, first-out (FIFO) fashion, similarly to a line of waiting people, who are served in first come, first served order. A `vector` is not a good implementation of a queue because inserting at the front causes all entries to shuffle along. `lists`, however, are good candidates for queuing. You add an item to a queue by using `push_front()`, and you take an item off the end by using `pop_back()`. Queues are commonly used in data communications, where you can have data coming in faster than it can be processed. So incoming data is *buffered*—that is, kept in a queue until it is used or the buffer *overflows*. The good thing about a `list` is that it never overflows, although it can *underflow*, when someone tries to take something off an empty queue; therefore, it is important to check size. Graphical user interface systems such as Windows typically maintain a *message queue*, which contains all the user's input. So it is possible to type faster than a program can process keystrokes.

Iterators

Containers such as `lists` and `maps` do not behave like arrays, so you can't use a `for` loop to go through the elements in them. Likewise, because these containers are not accessible randomly, you cannot use a simple integer index. You can use iterators to refer to elements of a container.

Iterating through a Container

Each container type has a distinct iterator associated with it. For instance, this is how you declare the iterator for `list<int>`:

```
;> list<int>::iterator ili;
```

You have previously seen the operator `::` in two forms: as the global scope operator (where it has one operand) and in the constant `ios::app`. Each container type has a scope that contains a type name (`iterator`), and the scope operator (`::`) allows you to access that type name. Again, using `typedef` can make for easier typing and understanding, as in the following example; remember that `ILI` is a completely different name from `ili` in C++:

EXAMPLE

```
;> typedef list<int> LI;
;> typedef LI::iterator ILI;
;> LI ls;  ls.push_back(1);  ls.push(2);
;> ILI ili = ls.begin();
;> *ili;
(int) 1
;> ++ili;
;> *ili;
(int) 2
;> for(ili = ls.begin(); ili != ls.end(); ++ili)
;1}  cout << *ili << endl;
1
2
```

In this example, the iterator `ili` is used for accessing the contents of the list `ls`. First, a list `ls` is created, and the numbers 1 and 2 are added to it. Then the iterator `ili` is declared and set to `ls.begin()`. The expression `*ili` gives you the first value in `ls`, and `++ili` moves the iterator to the next value. You use the *dereference* operator (`*`) to extract the value and the increment operator (`++`) to move to the next list item. (Note that `*` is used for both dereferencing and multiplication, in the same way that `-` is used for both `-2.3` and `2-3`. The *unary* and *binary* forms of the operator are quite different from one another.) The method `begin()` returns an iterator that points to the beginning of the list, but the method `end()` returns an iterator that is just beyond the end of the list. Once `++ili` has moved the iterator past the end, then `ili` becomes equal to `ls.end()`. Therefore, the `for` loop in the example visits each item in the list. This technique works for any list type, and it is a very common way of *iterating* over all items in a list. The vector—and in fact, any standard container—can also be traversed by using an iterator, as in the following example:

```
;> vector<string>::iterator vsi;
;> string tot;
;> for(vsi = vs.begin(); vsi != vs.end(); ++vsi) tot += *vsi;
```

In a case like this, you would use a plain `for` loop and use the vector as if it were an array, but being able to iterate over all containers like this allows you to write very general code that can be used with both `vectors` and `lists`.

Finding Items

Reinventing the wheel wastes your time and confuses those that follow you. The standard library provides a number of ready-to-use algorithms that do common tasks like searching and sorting. For instance, `find()` does a simple linear search for a value and returns an iterator that points to that value if it is successful. You specify the data to be searched by a pair of iterators. Assume that you have a list of strings, `ls`, which contains "john", "mary", and "alice":

EXAMPLE

```
;> list<string> iterator ii;
;> ii = find(ls.begin(), ls.end(), "mary");
;> *ii
(string&) 'mary'
;> *ii = "jane";
(string&) 'jane'
```

Note that `*ii` can be assigned a new value, too: It is a valid `lvalue` (short for left-hand value) and so can appear on the left-hand side of assignments. Because `*ii` is a reference to the second item in the list, modifying `*ii` actually changes the list. If `find()` does not succeed, it returns `ls.end()`.

You might wonder why you can't just pass the list directly to `find()`. The standard algorithms could do that but they prefer to work with *sequences*. Consider the following example of finding the second occurrence of 42 in a list of integers `li`. You use the result of the first `find()` as the start of the sequence for the second `find()`. We have to increment the iterator because it will be pointing to 42 after the first `find()`.

```
;> list<int>::iterator ili;
;> ili = find(li.begin(),li.end(),42); // first position
;> ++ili;                              // move past the '42'
;> ili = find(ili,li.end(),42);        // second position
;> *ili;
(int&) 42
```

`find()` accepts a sequence and not a container argument for another good reason: `find()` can work with ordinary arrays, which are not proper containers and have no size information. The following example illustrates this, with the standard algorithm `copy()`, which copies a sequence to a specified destination:

EXAMPLE

```
;> int tbl[] = {6,2,5,1};
;> int cpy[4];
;> copy(tbl,tbl+4,cpy);
;> show_arr(cpy,4);
6 2 5 1
;> int array[20];
;> copy(li.begin(), li.end(), array);
;> *find(tbl,tbl+4,5) = 42;
;> show_arr(tbl,4);
6 2 42 1
```

Note in this example how you specify the end of an array. `copy()` is very useful for moving elements from one type of container to another. For example, in this example, you move a list of integers into an array. Again, it is important that the array be big enough for the whole list; otherwise, you could unintentionally damage program memory. Likewise, you can use `find()`. The call returns a reference to the third element of `tbl`, which is then changed to 42.

If you have a sorted array-like sequence, then using `binary_search()` is much faster than `find()`, as discussed previously. A binary search requires a random access sequence, so it does not work for lists. Maps already have their own `find()` method, and the generic `find()` won't work on them.

Erasing and Inserting

Both `vectors` and `lists` allow you to erase and insert items, although these operations are faster with `lists` than with `vectors`. You specify a position using an iterator, such as the iterator returned by `find()`. Insertion occurs just before the specified position, as in the following example:

EXAMPLE

```
;> list<string> ls;
;> list<string>::iterator ils;
;> ls.push_back("one");
;> ls.insert(ls.end(),"two");       // definition of push_back()!
;> ls.insert(ls.begin(),"zero");    // definition of push_front()!
;> ils = find(ls.begin(),ls.end(),"two");
;> ls.insert(ils, "one-and-a-half");
;> while (ls.size() > 0) {
;1}   cout << ls.back() << ' ';
;1}   ls.pop_back(); // emptying the list in reverse
;1} }
two one-and-a-half one zero
```

`vectors` have methods for inserting and erasing elements, and they all involve moving elements. To erase the second element in a `vector` (remember that we are counting from zero and `begin()` refers to the first element), you would use code like this:

```
;> vi.erase(vi.begin() + 1);
;> vi.erase(vi.end() -  1);   // same as pop_back() !
```

Case Study: Calculating Simple Statistics

The case studies throughout this book show C++ being used for practical purposes. In Chapter 2 you saw that the library functions provided are biased toward scientific and engineering applications, but it's not difficult to write your own functions for statistical analysis. The same kind of data analysis that you would use on meteorological records can be used to look at stock prices, because they both involve values that vary over time. In the following two case studies I particularly want to show you how the standard algorithms can make processing data easier. Appendix B, "A Short Library Reference," will give you more information about the algorithms available.

In this first case study, we will download historical data that gives the performance of the S&P 500 over the past year, using this format: the date, the name of the stock or ticker symbol, the opening price, the highest price in the day, the lowest price in the day, the final price, and the trading volume. The date is in the same order as the ISO standard, but without the hyphens. Everything is separated by commas, which is convenient for spreadsheets, although not for program input /output. All of the stock data is bundled together in one big 5MB file, and so extracting data for a particular ticker symbol is going to be necessary. I have put some sample data on the accompanying CD-ROM, but you can get up-to-date historical data for the last year from `http://biz.swcp.com/stocks/`. (Click on the 'Get Full Set' button). The C++ source code for this case study will be found in `chap3\stats.cpp`, and the full year set of data is called `SP500HST.TXT`.

```
20000710,ABBA,67.75,70.25,67.6875,68.25,38349
```

The first task is to extract the stock of interest. Along the way, you will replace all commas with spaces. Although this is not a difficult operation to write, it has already been done for you to save some time. The `replace()` standard algorithm goes through any sequence, performing the required replacement. As you saw earlier in the chapter, strings are array-like, and although strings are not full containers, the basic algorithms will still work on them. Here is `replace()` in action:

EXAMPLE

```
;> int a[] = {1,2,0,23,0};
;> show_arr(a,5);
1 2 0 23 0
;> replace(a,a+5,0,-1);
;> show_arr(a,5);
1 2 -1 23 -1
;> string s = "a line from a song";
;> replace(s.begin(),s.end(),' ','-');
;> s;
(string) s = 'a-line-from-a-song'
```

Now that you have seen `replace()` in action, I can show you the function `extract_stock()`, which is given the stock ticker symbol. The first part reads each line until the line contains the symbol, or it runs out of data. If the symbol was found, then the second part reads each line, replaces commas with spaces, and writes the line out to another file. This continues as long as the line contains the ticker symbol.

```
bool extract_stock(string ofile,string stock)
{
  ifstream in(SFILE.c_str());
  string line;
  do {
     getline(in,line);
  }
  while (! in.eof() && line.find(stock) == string::npos);

  if (! in.eof()) { // found our stock!
    ofstream out(ofile.c_str());
    do {
       replace(line.begin(),line.end(),',',' ');
       out << line << endl;
       getline(in,line);
    } while (! in.eof() && line.find(stock) != string::npos);
    return true;
  } else
  return false;
}
;> extract_stock("yum.txt","YUM");
(bool) true
```

There is now a file `YUM.TXT` containing the S&P 500 data for the symbol YUM, without commas. (This will take a few seconds.) Next, you can easily read the values into some `vectors`; you don't know precisely how many trading days there were in the last 12-month period, so using `push_back()` is useful:

```
typedef vector<double> V;
typedef V::iterator IV;

bool read_any_stock(string data_file, V& oprices,
        V& lprices, V& hprices, V& fprices, V& volumes)
{
 ifstream in;
 if (! in.open(data_file.c_str())) return false;
 double lprice,hprice,fprice,vol,f;
 string date,stock;
 while (in >> date >> stock >> oprice >> lprice >> hprice >> fprice >> vol) {
   oprices.push_back(oprice);
   lprices.push_back(lprice);
   hprices.push_back(hprice);
   fprices.push_back(fprice);
   volumes.push_back(vol);
 }
 return true;
}

V open_price, low_price, high_price, final_price, volume;

bool read_stock(string dfile)
{
  low_price.clear();
  high_price.clear();
  final_price.clear();
  volume.clear();
  return read_any_stock(dfile,
           low_price,high_price,final_price,volume);
}
```

`read_any_stock()` is awkward to call, so you can define a helper function `read_stock()` that reads the values into global variables. Now the full year's prices are available for YUM; the first question is what the minimum and maximum prices have been.

The standard algorithms `max_element()` and `min_element()` save you the trouble of writing yet another loop to find the minimum and maximum values. These are not difficult operations to code, but they've already been done for your convenience. Note that these algorithms return an iterator that refers to the value and that the dereference operator (`*`) is needed to get the actual value.

```
;> read_stock("YUM.txt");
(bool) true
;> IV i1 = low_price.begin(), i2 = low_price.end();
*min_element(i1,i2);
(double&) 23.875
 *max_element(i1,i2);
(double&) 47.64
```

The most basic statistic is a plain average value, and it is made easy by the `accumulate()` algorithm, which gives you the sum of the elements of a sequence. To get the average value, you simply need to divide this sum by the number of values:

```
;> double val = 0;
;> accumulate(i1,i2,val)/low_price.size();
```

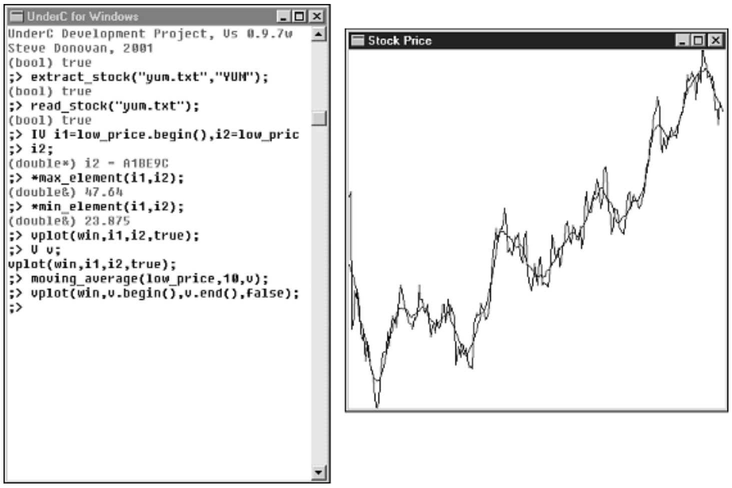
```
(double) 56.4577
;>
```

For time-series data like this, a plain average isn't very useful. Analysts are fond of *moving averages*, which smooth out the spikes and make the trends clearer. The moving average at any point is the average of the values of the neighboring points. The function `moving_average()` is passed an input, an output `vector<double>`, and a *smoothing width*. It works as follows: Define an interval with this width (sometimes called a *smoothing window*) and get the average value. Now move the interval along by one element and repeat. The series of average values generated by this moving interval is the moving average. Note how the interval is specified for `accumulate()`; The only thing to be careful about is how to handle the interval at both ends of the `vector`. Here I have used `max()` and `min()` to force the interval bounds to lie between `0` and `n-1`.

```
void moving_average(const V& vin, int width, V& vout)
{
  int w2 = width/2, n = vin.size();
  IV vstart = vin.begin();
  vout.resize(n);
  for(int i = 0; i < n; i++) {
     int lower = max(0,  i-w2);
     int upper = min(n-1,i+w2);
     double val = 0;
     val = accumulate(vstart+lower,vstart+upper,val);
     vout[i] = val/(upper - lower);
  }
}

;> V v;
;> moving_average(low_price,10,v);
;> vplot(win,low_price.begin(),low_price.end(),true);
;> vplot(win,v.begin(),v.end(),false);
```

This example includes some plot-generating calls in `stats.cpp` because sometimes a picture is worth a thousand words (or is that 4KB?). The averaged vector is indeed much smoother than the raw data (see Figure 3.1). The source code for `vplot()` is in `vplot.cpp`; the first call is passed a Boolean argument of `true` to force `vplot()` to scale to the data; any subsquent calls would pass `false` so it will reuse the scaling.



```
UnderC Development Project, Vs 0.9.7w
Steve Donovan, 2001
(bool) true
;> extract_stock("yum.txt","YUM");
(bool) true
;> read_stock("yum.txt");
(bool) true
;> IV i1=low_price.begin(),i2=low_pric
;> i2;
(double*) i2 = A1BE9C
;> *max_element(i1,i2);
(double&) 47.64
;> *min_element(i1,i2);
(double&) 23.875
;> vplot(win,i1,i2,true);
;> V v;
vplot(win,i1,i2,true);
;> moving_average(low_price,10,v);
;> vplot(win,v.begin(),v.end(),false);
;>
```

Figure 3.1: *The moving image is much smoother than the raw data.*

Case Study: Histograms

Suppose that you are presented with a large body of text. Is it possible to tell interesting things about the authors from the typical length of their sentences, their words, and so on? Some literary detectives think so; in this case study, your job is to collect these statistics and present them as a bar graph, otherwise known as a histogram. You already have one tool for the job—C++ input streams, which read in strings separated by spaces. C++ input streams are well suited to this task because they are not line based; text is naturally parsed into *tokens*, which are usually (but not always) words. For example, `"He said, let's go."` is read as `{"He" "said," "let's" "go."}`. So when you read in a token, you need to look at the end of each word for punctuation. The basic strategy is to read each word and punctuation character and to increment word and sentence frequency tables. These tables don't have to be very long, so arrays will work well.

First, the arrays need to be initialized to zero (never assume that this is already true!). The standard algorithm `fill_n()` is easier to use in this case than an explicit `for` loop:

```
const int MAXWORD = 30, MAXSENTENCE = 120;

int word_lengths[MAXWORD];
int sentence_lengths[MAXSENTENCE];

void init_arrays()
{
 fill_n(word_lengths,MAXWORD,0);
 fill_n(sentence_lengths,MAXSENTENCE,0);
}
```

Here is `collect_file_stats()`, which reads each word of the text using >>. Then it looks at the last character of the word, which is `word[len-1]` because strings are like arrays. If this character is punctuation, it must be removed from the string. The word length frequency table can be updated with a single statement: `++word_lengths[len]`. The next few lines count words in sentences.

```
bool collect_file_stats(string file, int word_lengths[],
                        int sentence_lengths[])
 {
  ifstream in(file.c_str());
  if (! in) return false; // sorry, can't open file!
  int wcount = 0;  // will count no. of words in each sentence
  string word;
  while (in >> word) {
    int len = word.length();
    char ech = word[len-1];   // last character of word
    if (ech == '.' || ech == ',' || ech == ';') {
       word = word.substr(0,len-1); // chop off punctuation
       —len;
    }
    ++word_lengths[len];
    ++wcount;
    if (ech == '.') {
      ++sentence_lengths[wcount];
      wcount = 0;
    }
  } // end of while(...)
 } // end of collect_file_stats
;> init_arrays();
;> collect_file_stats("chap1.txt",word_lengths,sentence_lengths);
;> show_arr(word_lengths,15);
0 266 854 866 806 472 312 316 232 106 104 56 62 30 4
;> show_arr(sentence_lengths,15);
0 4 0 10 0 4 12 8 6 6 4 4 6 4 8
```

The result of using this code is two frequency tables, `word_lengths` and `sentence_lengths`. `word_lengths[1]` shows the number of words with one character, `word_lengths[2]` shows the number of words with two characters, up to about 15 or so characters. I have used the useful function `show_arr()` from earlier in the chapter to show you the values when the first chapter of *The Hound of the Baskervilles* is analyzed.

Two Approaches to Histograms

The easiest way to print out a histogram is to print it on its side. You can fill strings or character arrays with a chosen character, but the easiest method is just to write a character out to `cout`. (You can make the bars thicker by nesting another loop inside the `i` loop.)

EXAMPLE

```
void histo_1 (int values[], int sz)
{
 for(int i = 0; i < sz; i++) {
   for(int k = 0; k < values[i]; k++) cout << '*';
   cout << endl;
 }
}
;> histo_1(sentence_lengths,10);
0
1 ****
2
3 **********
4
5 ****
6 ************
7 ********
8 ******
9 ******
```

Note that this method is just not going to work with `word_lengths`, since the values are very large. These values need to be *scaled* by choosing a maximum value for the display (say 60 characters across) and generating an array within those bounds. The maximum value of the input data is found as before, and the input data is multiplied by the scaling factor, which is just the ratio of the desired maximum value to the actual maximum value.

EXAMPLE

```
void scale_data(int in_data[], int out_data[], int sz, int imax)
{
  int maxval = *max_element(in_data,in_data+sz);
  for(int i = 0; i < sz; i++)
     out_data[i] = (imax*in_data[i])/maxval;
}

;> int scaled_lengths[16];
;> scale_data(word_lengths,scaled_lengths,15,60);
;> show_arr(scaled_lengths,15);
0 18 59 60 55 32 21 21 16 7 7 3 4 2 0
;> histo_1(scaled_lengths,15);
```

OUTPUT

```
0
1 ******************
2 ***********************************************************
3 ************************************************************
4 *******************************************************
5 ********************************
6 *********************
7 *********************
8 ****************
9 *******
10 *******
11 ***
12 ****
13 **
14
;>
```

How can you create a histogram that is upright? You start with the maximum value as a level, and then you run along the array, comparing values to the level. If the values are greater than or equal to the level, you write out a string, and if they are less than the level, you write out spaces:

```
void histo_2 (int values[], int sz)
{
  int level = max_arr(values,sz);   // std method to do this?
  for(; level > 0; level—) {      // leave out 1st part of for-loop
    for(int i = 0; i < sz; i++)
      if (values[i] >= level) cout << "**** ";
                        else cout << "     ";
    cout << endl;
\  }
}
;> scale_data(word_lengths,scaled_lengths,10,20);
;> histo_2(scaled_lengths,10);
```

```
               ****
          **** ****
          **** **** ****
          **** **** ****
          **** **** ****
          **** **** ****
          **** **** ****
          **** **** ****
          **** **** ****
          **** **** ****
          **** **** **** ****
          **** **** **** ****
          **** **** **** ****
          **** **** **** **** **** ****
     **** **** **** **** **** **** ****
     **** **** **** **** **** **** **** ****
     **** **** **** **** **** **** **** ****
     **** **** **** **** **** **** **** ****
     **** **** **** **** **** **** **** **** ****
     **** **** **** **** **** **** **** **** ****
**** **** **** **** **** **** **** **** **** ****
```

This example hardly exploits the supercharged graphics of modern machines, of course. You can also create histograms by using the Turtle Graphics interface of UnderC for Windows. Appendix B, "A Short Library Reference" gives you all the information you need about using Turtle Graphics for now. I will be discussing it further in Chapter 4, "Programs and Libraries."

What's Next

In this chapter, you saw how C++ arrays and standard containers can be used to organize tables of data. C++ also supplies a set of convenient algorithms that saves you from having to write your own. Up to now, you have used C++ interactively with UnderC by writing functions; you are now ready to write and compile programs using GCC. I will explain precisely what is meant by a program library, look at how namespaces help you to organize your programs, and finally how to write robust code using exceptions.

4

Programs and Libraries

In the previous chapters, you have learned about the basic elements of C++: variables, expressions, arrays, control structures, and functions. To this point you have not written a program, and I've taken this approach deliberately. The usual approach would be to show you a program and promise to explain it later. Instead, using C++ interactively with UnderC allows you to play with functions and expressions directly. When you are familiar with functions and expressions, you can easily write real programs.

In this chapter you will learn

- How to include header files and prototype functions
- How to use namespaces to separate functions into families
- How to compile and run a simple program with command-line arguments
- How to bullet-proof programs by using exceptions

Header Files

All functions and variables in C++ need to be declared before they can be used. C++ provides operators, constants, and a few basic types—and that's that. So where are functions like `sin()` declared? Where does the `string` type come from? These declarations are in the standard header files. You have not needed to know this because UnderC includes the chief header files automatically. In your project directory, take a look at a file called `defs.h`, which usually contains these statements:

```
#include <iostream>
#include <fstream>
#include <stringstream>
#include <string>
#include <vector>
#include <list>
#include <map>
using namespace std;
```

The `#include` statements don't look like usual C++ statements, and in fact they are more like UnderC commands like `#l` than like C++ statements. An `#include` statement must stand on a line by itself, and it does not include a semicolon. Like `#l`, `#include` directly includes a file into the program text that the compiler sees. `#include` is a *preprocessor directive*, which is a command that controls how program text is initially processed; the *include directive* allows you to load files that contain declarations. These standard include files are usually named for the type they define (for example, `<string>`), except for `<iostream>`, which declares the input/output streams, and `<fstream>`, which declares the file streams (`ofstream` and `ifstream`). The angle brackets (`<>`) cause the preprocessor to look for these files in a special `include` directory. In the case of UnderC, if `\UCW` is the directory where `UCW.EXE` is kept, then `\UCW\include` contains these headers.

The final statement in `defs.h`, `using namespace std`, is a proper C++ statement, and you will learn more about it later in this chapter, in the section "Namespaces."

Libraries

C++ has a reputation for being a big language, and indeed it is very sophisticated. However, C++ executables can be very small; by far the greatest amount of code in any program is contained in the language's *libraries*. A C++ library is a collection of functions and types that have already been compiled into machine code. Using libraries means you don't have to do things like write a square-root function, or speak directly to the PC's BIOS to get characters out on a screen. These types of things have already been done for you and tested by legions of obsessive programmers. The standard libraries come with the language and are straightforward to include in your programs; usually all you have to do is include their header files. In fact, the C++ libraries are what make C++ such a powerful tool.

If the standard libraries don't have what you are looking for, then usually somebody out there has had the same problem and has posted the code you need on the Internet. Programmers build up tool chests of useful functions and types; they rarely write nontrivial programs without reusing some libraries they developed previously. Although UnderC is not the fastest implementation of C++ around (because it is an interpreter), it is straightforward to use *dynamic link libraries* (DLLs), which you can build by using a compiler such as GCC. (Appendix D, "Compiling C++ Programs and DLLs with GCC and BCC32," has the details.)

The libraries used by a working C++ program (that is, the *runtime* libraries) consist of hundreds of thousands of lines of C++, C, and assembly code. Progress would be very slow if all this code had to be compiled each time (a simple "Hello, World" program would take minutes to build), so libraries are available as *object code*, which is *linked* into the program. That is, the linker examines your program code and includes any extra functions that have been refered to, such as `sqrt()` or `sin()`. Nontrivial C++ programs consist of many separate files that are separately compiled. People don't use `#include` with the actual source of functions; rather, they use *prototypes*—function declarations that tell the compiler what arguments a function takes and what type it returns, but don't give a full definition of their code. Instead of a code block (`{}`), a prototype has a simple semicolon. Prototypes or function declarations are promises that the function will be defined at some later stage. Eventually, of course, you need to define the function fully. For example, UnderC complains if a function is prototyped but not yet defined. Using prototypes means that you can write functions in the most logical order. This example shows a function `repeat()`, which is declared first as a prototype, but not defined. You cannot call it directly at this point, but you may define other functions such as `repeat2()`, which refer to `repeat()` before it is fully defined.

```
;> string repeat(string s, int I);
;> repeat("help",1);
'repeat' is not defined yet <temp>
;> string repeat2(string s)
;> { return repeat(s,2); }     // no error!
;> string repeat(string s, int I) // now define repeat()
;> { string res; for(int k=0;k<I;k++) res += s; return res; }
;> repeat2("help");
(string) 'helphelp'
```

In a traditional C++ system, the function is resolved by the *linker*, which looks at the functions declared in each file and finds their definitions in the object files, either from other code in the project or from libraries.

Classifying Characters

To see examples of some more of the facilities supplied by the C++ runtime library, we'll look at the problem of classifying characters. For instance, is a character alphabetic? We know characters are stored as integers, and we assume that they are stored in order. Here I have defined a function `is_alpha()`, which uses a simple test: if the character is greater or equal to 'A', and less or equal to 'Z', then it must be an uppercase alphabetic character:

EXAMPLE

```
;> bool is_alpha(char ch)
;>  { return 'A' <= ch && ch <= 'Z'; }
;> is_alpha('A');
(bool) true
;> (int)'a';
(int) 97
;> (int)'A';
(int) 65
```

This example works well, but only for uppercase letters. You might assume that "a" is less than "A", but that is not the case, as you can see. Generally, there are too many assumptions involved with comparing characters, and most of them are true only for ASCII–Unicode works completely differently. Therefore, it is a good idea to always use the C++ classification functions, which return nonzero integer values if they are true. To use them, you must include the `<cctype>` standard header, which C programmers know better as `<ctype.h>`. Here I show `isalpha()` and `isdigit()` in action:

```
;> #include <cctype>
;> isalpha('x');
(int) 2
;> isdigit('x');
(int) 0
;> isdigit('9');
(int) 4
```

The values returned by these functions are not important; we simply need to know that the returned values are nonzero, which will translate as being `true` in C++. Another useful function is `isspace()`, which is true if the character represents whitespace (that is, spaces, tabs, line feeds, and so on). The following is an example of its use in implementing a `trim()` function that operates on strings and removes leading and trailing spaces. That is, " the dog " becomes "the dog". Its BASIC equivalent is TRIM$:

EXAMPLE

```
string trim(string s) {
   int k = 0, n = s.length();
   while (k < n && isspace(s[k])) ++k;   // go forward to first nonspace in s
   if (k == n) return "";                // there were none!
   int ke = n-1;                         // position of last character
   while (isspace(s[ke])) —ke;           // go backwards to last nonspace
   return s.substr(k,ke-k+1);
 }
;> trim("  happy dog    ");
(string) 'happy dog'
;> trim("\tone\ttwo\t\n");
(string) 'one   two'
```

If you know the C++ library well, then you won't waste your time rewriting it, and other programmers will find your code easier to read.

Programs

A *program* is a collection of functions. One of those functions must have the name `main()`. When program execution begins, `main()` will be called automatically, together with any parameters. The function `main()` must return an `int`.

The Function `main()`

After `main()` has been defined, the UnderC command `#r` (that is, run) causes a new console window to be created, showing the output generated by the program (see Figure 4.1), as in the following example:

```
;> int main() {
;1} cout << "Hello World\n";
;1} return 0;
;1} }
;> #r
```

If `main()` is not defined, `#r` produces a "cannot find `'main'`" error. The function `main()` is called the *entry point* of the program, and it should return a value, unlike in C. C++ compilers (like GCC, but not UnderC) are usually tolerant about returning a value from `main()`, but it's good to get into the habit of writing `main()` as a proper function.

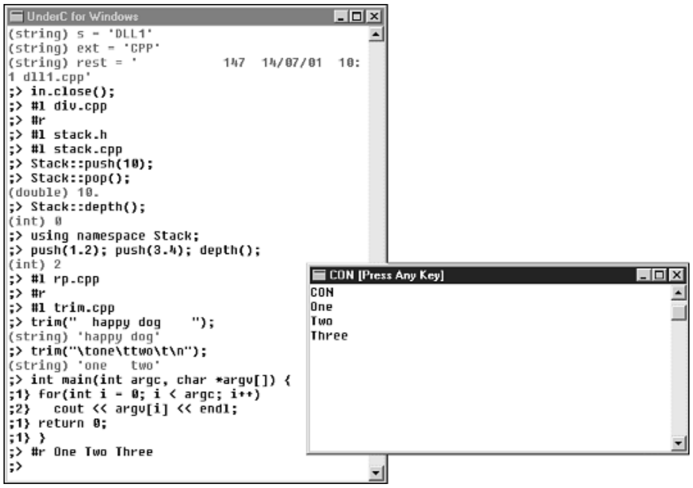
`main()` can optionally have arguments, like this. The arguments follow the run command in UnderC:

EXAMPLE

```
;> int main(int argc, char *argv[]) {
;1}  for(int i=0; i < argc; i++)
;2}    cout << argv[i] << endl;
;1}  return 0;
;1} }
;> #r One Two Three
```

OUTPUT

```
CON
One
Two
Three
```



```
(string) s = 'DLL1'
(string) ext = 'CPP'
(string) rest = '          147   14/07/01   10:
1 dll1.cpp'
;> in.close();
;> #l div.cpp
;> #r
;> #l stack.h
;> #l stack.cpp
;> Stack::push(10);
;> Stack::pop();
(double) 10.
;> Stack::depth();
(int) 0
;> using namespace Stack;
;> push(1.2); push(3.4); depth();
(int) 2
;> #l rp.cpp
;> #r
;> #l trim.cpp
;> trim("  happy dog    ");
(string) 'happy dog'
;> trim("\tone\ttwo\t\n");
(string) 'one   two'
;> int main(int argc, char *argv[]) {
;1} for(int i = 0; i < argc; i++)
;2}   cout << argv[i] << endl;
;1} return 0;
;1} }
;> #r One Two Three
;>
```

```
CON
One
Two
Three
```

Figure 4.1: *An example of the output generated by the program.*

The second argument of `main()`, `argv`, is basically an array of C-style strings. Notice that `argv[0]` is not the first argument passed; the command-line arguments are `argv[1]` to `argv[argc-1]`. The first value, `argv[0]`, is the name of the file that contained the `main()` function (in this case, just `CON`, the console).

Building Your First Program

It's finally time to compile and run a complete C++ program. Before the code shown in the previous section will compile correctly, it needs the `<iostream>` header for the output stream. Save the following code as `args.cpp` using your favorite text editor, load it, and run the program:

```
// args.cpp
#include <iostream>
using namespace std;
int main(int argc, char *argv[])
{
  for(int i=0; i < argc; i++)
    cout << argv[i] << endl;
  return 0;
}
```

If you installed the software from the CD, you should have a fully functioning version of the GNU C++ compiler on your system. The compiler's name is `c++`, and here is the result of compiling and running `args.cpp` on my system, at the command prompt (user input is bold):

EXAMPLE

```
C:\bolsh\ucw\chap4>c++ args.cpp
C:\bolsh\ucw\chap4>a One Two Three
C:\BOLSH\UCW\CHAP4\A.EXE
One
Two
Three
```

The compiler will work for a second or two, and the resulting compiled program will be called `a.exe`. (This name is an old Unix tradition.) You then run `a.exe` with three command-line arguments, and you get the expected output, except now `argv[0]` is the full path to the program. An interesting property of this program is that it does *wildcard expansion*: Any command-line arguments that contain wildcard characters (such as `*`) are expanded into a full list of files matching those wildcards. You can use this simple little program to list all the `.cpp` files you have in this directory:

```
C:\bolsh\ucw\chap4>a *.cpp
C:\BOLSH\UCW\CHAP4\A.EXE
args.cpp
basic.cpp
dll1.cpp
stack.cpp
```

NOTE

UnderC currently doesn't do wildcard expansion. The Borland compiler does not include this facility by default, but you can specify it as an option. See Appendix D for details on how to use other compilers.

Separate Compilation

Programmers often seem to be obsessed with the number of lines of code in their programs. Unfortunately, Windows doesn't come standard with an `lc` (for line count) command, but you can write a small function to count the number of lines in an input stream:

EXAMPLE

```
// count.cpp
#include <iostream>
using namespace std;

const int LINESIZE = 512;

int count_lines(istream& in)
{
 char buff[LINESIZE];
 int k = 0;
 while (in.getline(buff,LINESIZE)) ++k;
 return k;
}

;> #l count.cpp
;> ifstream ifs("args.cpp");
;> count_lines(ifs);
(int) 9
```

Note that this example does not pass `count_lines()` a file, but a reference (`istream&`) to an input stream. If you are not particularly interested in the text for each line, you can use the `getline()` method that quickly fetches the line into an array of characters. You can then compile `count.cpp` directly, but with the `-c` command-line flag, which means "don't attempt to link." If you leave out this flag, the linker complains that it cannot find `main()`. The output is an object file `count.o`, which is the machine code. (If you use the Borland or Microsoft compilers, this file has a different extension: `.obj`. Object files from different compilers are in general not compatible.) Here I have compiled `count.cpp` into an object file and shown the resulting output file `count.o` (which will not be ASCII):

```
C:\bolsh\ucw\chap4>c++ -c count.cpp
C:\bolsh\ucw\chap4>dir count.o

 Volume in drive C has no label
 Volume Serial Number is 11F3-3136
 Directory of C:\bolsh\ucw\chap4

COUNT    O           4,629  26/07/01  12:59 count.o
         1 file(s)          4,629 bytes
         0 dir(s)     733,261,824 bytes free
```

You can compile and link this with a file that contains a `main()` function to produce an executable:

EXAMPLE

```
// main1.cpp
#include <iostream>
#include <fstream>
using namespace std;

int count_lines(istream& in);  // definition in count.cpp!

int main(int argc, char *argv[])
{
  ifstream in(argv[1]);
  cout << "lines " << count_lines(in) << endl;
  return 0;
}
C:\bolsh\ucw\chap4>c++ main1.cpp count.o
C:\bolsh\ucw\chap4>a main1.cpp
lines 12
```

You need `lc` to operate on standard input if there are no files and on all the supplied files. It is fairly easy to write a more sophisticated `main()` function that does this. Note that you can pass both the standard input stream (`cin`) and a file stream (`ifstream` object) to `count_lines()` because they are both `istream`s. In compiling this version I've used the `-o` option to name the output file `lc.exe` rather than `a.exe`:

EXAMPLE

```
// main2.cpp
#include <iostream>
#include <fstream>
using namespace std;

int count_lines(istream& in);  // definition in count.cpp!

int main(int argc, char *argv[])
{
 int lcount = 0;
 if (argc == 1)
   lcount = count_lines(cin);
 else for(int i = 1; i < argc; i++) {
   ifstream in(argv[i]);
   if (!in) cerr << "Can't open '" << argv[i] << "'\n";
   else lcount += count_lines(in);
 }
 cout << lcount << " lines\n";
 return 0;
}
C:\bolsh\ucw\chap4>c++ -o lc.exe main2.cpp count.o
C:\bolsh\ucw\chap4>dir | lc
52 lines
C:\bolsh\ucw\chap4>lc *.cpp *.h
289 lines
```

In the first command, the output of `dir` is 'piped' to `lc` as its standard input, which is then counted. Because the second command is given wildcard parameters, `lc` is actually passed a list of all `.cpp` and `.h` files.

In preparing the second version of `lc`, you don't have to recompile `count.cpp`, which shows the advantage of separate compilation. In a large project there may be dozens of files, each with many functions. Recompiling all these files would take minutes or even hours. But deciding what needs to be recompiled can be tricky (remember that if the include files change, the source file must also be recompiled). You need to either create a project in an integrated development environment (IDE) such as Quincy or write a make file. Appendix D, “Compiling C++ Programs and DLLs with GCC and BCC32,” describes this process in detail.

Usually, for each source file there is an include file (otherwise known as a header file) that has prototypes for all functions that need to be exported from that file. In this case it would be very short, because `count.cpp` only exports one function.

Namespaces

A large program may have thousands of functions, and it can become difficult to tell them apart. No matter how well you name them, they will get mixed up. This is, of course, not just a problem with programs; the phone book of even a small town would be impossible to use if people did not have surnames. Programmers often use naming conventions to keep functions separate (for example, stack operations would be called `stack_pop()`, `stack_push()`, and so on) but are not always consistent. Namespaces are a way of grouping functions and variables together, in much the same way that families share the same surname.

The Global Namespace

A global declaration is visible throughout a program. Although they are not necessarily evil, global variables can be a problem because distant parts of the program become coupled together. Global side effects make it possible for subsystems to interfere with each other in ways that are not obvious. You could have two files, neither of which call each other but that still interact because they operate on the same global data. Another problem is if you have thousands of global declarations, they will start ‘colliding’ with each; it becomes hard to come up with a unique meaningful name that isn’t less than 20 characters long. The set of all global symbols is called the global namespace.

Therefore, you generally want to keep data private within files. But even if they are not declared globally, variables declared at the file scope can interfere with each other. Here is a simple (and pointless) program that consists of two files, which both contain the names `jo` and `fred`:

```
// one.cpp
int fred = 999;
int jo()   { return fred; }
int main() { return jo(); }
```

```
// two.cpp
int fred = 648;
int jo()   { return fred; }
```

```
C:\bolsh\ucw\chap4>c++ one.cpp two.cpp
C:\..\..o(.text+0x0):two.cpp: multiple definition of 'jo(void)'
C:\..\8xfb.o(.text+0x0):one.cpp: first defined here
C:\..\8i2hfb.o(.data+0x0):two.cpp: multiple definition of 'fred'
C:\..\8xfb.o(.data+0x0):one.cpp: first defined here
```

Each file is compiled correctly, but the linker complains of multiple definitions. It is a very good thing that the compile failed; I can attest from personal experience that this is one hard fault to debug! It is useful to insist that variables and functions be genuinely local to a file. You can do this by using a *nameless namespace* (or an *anonymous namespace*), as in the following example, which shows `two.cpp` again:

EXAMPLE

```
// two.cpp
namespace {
  int fred = 648;

  int jo() { return fred; }
}
```

Now when `one.cpp` and `two.cpp` are compiled and linked, there is no error: `fred` and `jo()` are completely private to `two.cpp`. The ideal situation is for each file to be as independent as possible. (C programmers used to do this with the `static` attribute, but this is now officially considered old-fashioned and "depreciated," which means the International Standard Organization [ISO] C++ standards committee wishes it would go away.)

Keeping Similar Functions Together in a Namespace

Say you are using a module to implement a stack. The implemention is easily done with an array and an index, but it is a good idea to provide a set of functions that manipulate the representation. There are two operations that modify the stack: `push()`, which puts a new value onto the stack, and `pop()`, which takes a value off the stack. It is also necessary to know how many items have been put on the stack: `depth()` and `empty()`.

That is, it's not generally a good idea to write `arr[iptr++]=val` every time you want to push something onto the stack. It's easy to get it wrong (I usually got confused between `++iptr` and `iptr++`), and perhaps one day we will want to implement the stack using a list.

You can then use an explicit namespace to group the functions together. The following example has an *implementation file* `stack.cpp` and an *interface file* `stack.h`:

EXAMPLE

```
// stack.cpp
namespace {  // private to this module
  const int STACKSIZE = 100;
  double arr[100];
  int iptr = 0;
}
namespace Stack {
  void   push(double v) { arr[iptr++] = v;    }
  double pop()          { return arr[--iptr]; }
  int    depth()        { return iptr;        }
  bool   empty()        { return depth()==0;  }
}

//stack.h
namespace Stack {
  void   push(double v);
  double pop();
  int    depth();
  bool   empty();
}
```

After loading this file, you can access its functions by giving the *fully qualified name,* by using the scope operator (that is, `::`). Notice that the function `empty()` is defined in terms of `depth`(), which doesn't need to be qualified because it belongs to the same family (that is, it is within the same scope). If a family consisted of Jane Smith, John Smith, and Peter Smith, then it would be unnecessary (although not incorrect) for John to call Peter by his full name, Peter Smith. To use this module I must load the source into the system with #l, and I can then use the stack operations using their qualified names:

```
;> #l stack.cpp
;> Stack::push(1.2);
;> Stack::push(3.2);
;> Stack::depth();
(int) 2
;> Stack::pop();
(double) 3.2
```

You may find them irritating to type at first, but fully qualified names tell precisely where a function comes from, and they therefore make it easier to understand large programs. Remember that the purpose of namespaces is to organize programs better for humans (computers don't care about keeping track of thousands of names). A person reading your code should be able to tell where every name is defined.

You can make a whole namespace available globally by using the `using` directive; in this case, for example, you would say that the namespace `Stack` has been injected into the global namespace:

```
;> using namespace Stack;
;> push(3.4);
;> empty();
(bool) false
```

It's as if a whole family of functions is introduced to the system at once. This is particularly useful when you're working interactively.

The `std` Namespace

Just about every simple C++ program includes `using namespace std`:

```
...
namespace std {.
  ....
}
```

The entire standard library is defined inside `std`; that is, all the definitions within `<iostream>`, `<string>`, and so on are included in a `std` namespace definition.

An important feature of namespaces is that they are always open, which means they can always be added to. Each `std` namespace definition adds its functions and types to `std`. With C++, programmers can explicitly qualify each name from the standard library, or they can make the definitions globally available by using `using namespace std`. Here is a simple program in which each name is explicitly qualified with `std` using the scope operator (`::`):

EXAMPLE

```
// strict.cpp
#include <iostream>
#include <string>

int main() {
  std::string s;
  while (std::cin >> s) std::cout << s << std::endl;
  return 0;
}
```

I find this method a bit fussy and pedantic. After all, we know `string`, `cin`, `cout`, and `endl` well by now and have earned the right to call them by their first names. However, some people feel strongly that `using namespace std` causes namespace pollution because everything is dumped into the global namespace, which is what namespaces were designed to prevent. You need to understand the implications of `using namespace std`, and you need to recognize that there is one case where it is always a bad idea. The header file for the count.cpp file defined early has only one prototype, but it must include <iostream> for std::istream. That way, any programmer who uses count.h doesn't have to remember that <iostream> must be included before count.h. (You do not have to worry about the programmer then including <iostream> twice; it is simply ignored the second time.) This is a case where you don't just bring in the entire std namespace with using namespace std; it is considered particularly bad manners because it means programmers cannot make their own choices. Here is a well-mannered header file for count.cpp:

```
// count.h
#include <iostream>
int count(std::istream& is);
```

In an office context, surnames are necessary only if there are multiple people with the same first name. Keeping the standard definitions in std means that you can avoid problems with namespace collision. Say you have to use an old library that has defined its very own string type (writing your own string library was common before the standard library was developed) and called it string. In this case you would not inject std into the global namespace because then the two definitions of string would collide with each other. Using std::string makes it absolutely clear what kind of string you're working with. The same is true for simple, often-used names such as list, vector, and map. It is still tedious to write std::cout, but you can always use using declarations (as opposed to using directives), as in the following example. These actually make the name available in the global namespace:

```
using std::cout;
using std::endl;
using std::cerr;
```

Why can't you just change the headers for the library and avoid all the extra typing? You could then change each library's string to lib_string, for example. But this would not help, unless you actually had the libraries' source code (and with commercial libraries, you usually don't) because the function references in the object files still contain string, not lib_string.

You can of course add your own functions to std, but that would defeat the purpose of std, which is to collect together the standard library functions and structures. The std types all appear in lowercase, so it is helpful to use the naming convention of beginning types with an uppercase letter so that it is obvious what the types are.

Defensive Programming

There are five desirable properties of good programs: They should be robust, correct, maintainable, friendly, and efficient. Obviously, these properties can be prioritized in different orders, but generally, efficiency is less important than correctness; it is nearly always possible to optimize a well-designed program, whereas badly written ”lean and mean” code is often a disaster. (Donald Knuth, the algorithms guru, says that “premature optimization is the root of all evil.”)

Here I am mostly talking about programs that have to be used by non-expert users. (You can forgive programs you write for your own purposes when they behave badly: For example, many scientific number-crunching programs are like bad-tempered sports cars.) Being unbreakable is important for programs to be acceptable to users, and you, therefore, need to be a little paranoid and not assume that everything is going to work according to plan. ‘Defensive programming’ means writing programs that cope with all common errors. It means things like not assuming that a file exists, or not assuming that you can write to any file (think of a CD-ROM), or always checking for divide by zero.

In the next few sections I want to show you how to ‘bullet-proof’ programs. First, there is a silly example to illustrate the traditional approach (check everything), and then I will introduce exception handling.

Bullet-Proofing Programs

Say you have to teach a computer to wash its hair. The problem, of course, is that computers have no common sense about these matters: “Lather, rinse, repeat” would certainly lead to a house flooded with bubbles. So you divide the operation into simpler tasks, which return true or false, and check the result of each task before going on to the next one. For example, you can’t begin to wash your hair if you can’t get the top off the shampoo bottle.

Defensive programming means always checking whether an operation succeeded. So the following code is full of `if-else` statements, and if you were trying to do something more complicated than wash hair, the code would rapidly become very ugly indeed (and the code would soon scroll off the page):

EXAMPLE

```
void wash_hair()
{
  string msg = "";
  if (! find_shampoo() || ! open_shampoo()) msg = "no shampoo";
  else {
    if (! wet_hair()) msg = "no water!";
    else {
      if (! apply_shampoo()) msg = "shampoo application error";
      else {
        for(int i = 0; i < 2; i++)  // repeat twice
          if (! lather() || ! rinse()) {
                msg = "no hands!";
                break;  // break out of the loop
          }
          if (! dry_hair())  msg = "no towel!";
      }
    }
  }
  if (msg != "") cerr << "Hair error: " << msg << endl;
  // clean up after washing hair
  put_away_towel();
  put_away_shampoo();
}
```

Part of the hair-washing process is to clean up afterward (as anybody who has a roommate soon learns). This would be a problem for the following code, now assuming that `wash_hair()`returns a string:

```
string wash_hair()
{
 ...
  if (! wet_hair()) return "no water!"
  if (! Apply_shampoo()) return "application error!";
...
}
```

You would need another function to call this `wash_hair()`, write out the message (if the operation failed), and do the cleanup. This would still be an improvement over the first `wash_hair()` because the code doesn't have all those nested blocks.

NOTE

Some people disapprove of returning from a function from more than one place, but this is left over from the days when cleanup had to be done manually. C++ guarantees that any object is properly cleaned up, no matter from where you return (for instance, any open file objects are automatically closed). Besides, C++ exception handling works much like a `return`, except that it can occur from many functions deep. The following section describes this and explains why it makes error checking easier.

Catching Exceptions

An alternative to constantly checking for errors is to let the problem (for example, division by zero, access violation) occur and then use the C++

exception-handling mechanism to gracefully recover from the problem. *Exceptional* usually means out of the ordinary and unusually good, but when it comes to errors, the word has a more negative meaning. The system *throws* an exception when some error condition happens, and if you don't *catch* that exception, it will give you a dialog box that says something like "your program has caused an error——goodbye." You should avoid doing that to your users—at the very least you should give them a more reassuring and polite message.

If an exception occurs in a `try` block, the system tries to match the exception with one (or more) `catch` blocks.

```
try { // your code goes inside this block
  ... problem happens - system throws exception
}
catch(Exception) { // exception caught here
  ... handle the problem
}
```

It is an error to have a `try` without a `catch` and vice versa. The `ON ERROR` clause in Visual Basic achieves a similar goal, as do signals in C; they allow you to jump out of trouble to a place where you can deal with the problem. The example is a function `div()`, which does integer division. Instead of checking whether the divisor is zero, this code lets the division by zero happen but catches the exception. Any code within the `try` block can safely do integer division, without having to worry about the problem. I've also defined a function `bad_div()` that does not catch the exception, which will give a system error message when called:

EXAMPLE

```
int div(int i, int j)
{
 int k = 0;
 try {
   k = i/j;
   cout << "successful value " << k << endl;
 }
 catch(IntDivideByZero) {
   cout << "divide by zero\n";
 }
 return k;
}
;> int bad_div(int i,int j) { return i/j; }
;> bad_div(10,0);
integer division by zero <main> (2)
;> div(2,1);
successful value 1
(int) 1
```

```
;> div(1,0);
divide by zero
(int) 0
```

This example is not how you would normally organize things. A lowly function like `div()` should not have to decide how an error should be handled; its job is to do a straightforward calculation. Generally, it is not a good idea to directly output error information to `cout` or `cerr` because Windows graphical user interface programs typically don't do that kind of output. Fortunately, any function call, made from within a `try` block, that throws an exception will have that exception caught by the `catch` block. The following is a little program that calls the (trivial) `div()` function repeatedly but catches any divide-by-zero errors:

EXAMPLE

```
// div.cpp
#include <iostream>
#include <uc_except.h>
using namespace std;

int div(int i, int j)
{  return i/j;   }

int main() {
 int i,j,k;
 cout << "Enter 0 0 to exit\n";
 for(;;) { // loop forever
   try {
     cout << "Give two numbers: ";
     cin >> i >> j;
     if (i == 0 && j == 0) return 0; // exit program!
     int k = div(i,j);
     cout << "i/j = " << k << endl;
   } catch(IntDivideByZero) {
     cout << "divide by zero\n";
   }
  }
  return 0;
}
```

Notice two crucial things about this example: First, the error-handling code appears as a separate exceptional case, and second, the program does not crash due to divide-by-zero errors (instead, it politely tells the user about the problem and keeps going).

Note the inclusion of `<uc_except.h>`, which is a nonstandard extension specific to UnderC. The ISO standard does not specify any hardware error exceptions, mostly because not all platforms support them, and a standard has to work everywhere. So IntDivideByZero is not available on all systems. (I have included some library code that implements these hardware exceptions for GCC and BCC32; please see the Appendix for more details.)

How do you catch more than one kind of error? There may be more than one catch block after the try block, and the runtime system looks for the best match. In some ways, a catch block is like a function definition; you supply an argument, and you can name a parameter that should be passed as a reference. For example, in the following code, whatever do_something() does, catch_all_errors() catches it—specifically a divide-by-zero error—and it catches any other exceptions as well:

```
void catch_all_errors()
{
  try {
    do_something();
  }
  catch(IntDivideByZero) {
    cerr << "divide by zero\n";
  }
  catch(HardWareException& e) {
    cerr << "runtime error: " << e.what() << endl;
  }
  catch(Exception& e) {
    cerr << "other error " << e.what() << endl;
  }
}
```

The standard exceptions have a what() method, which gives more information about them. Order is important here. Exception includes HardwareException, so putting Exception first would catch just about everything. When an exception is thrown, the system picks the first catch block that would match that exception. The rule is to put the catch blocks in order of increasing generality.

Throwing Exceptions

You can throw your own exceptions, which can be of any type, including C++ strings. (In Chapter 8, “Inheritance and Virtual Methods,” you will see how you can create a hierarchy of errors, but for now, strings and integers will do fine.) It is a good idea to write an error-generating function fail(), which allows you to add extra error-tracking features later. The following example returns to the hair-washing algorithm and is even more paranoid about possible problems:

EXAMPLE

```
void fail(string msg)
{
  throw msg;
}

void wash_hair()
{
  try {
    if (! find_shampoo()) fail("no shampoo");
    if (! open_shampoo()) fail("can't open shampoo");
    if (! wet_hair())     fail("no water!");
    if (! apply_shampoo())fail("shampoo application error");
    for(int i = 0; i < 2; i++)  // repeat twice
      if (! lather() || ! rinse()) fail("no hands!");
    if (! dry_hair())     fail("no towel!");
  }
  catch(string err) {
    cerr << "Known Hair washing failure: " << err << endl;
  }
  catch(...) {
    cerr << "Catastropic failure\n";
  }
  // clean up after washing hair
  put_away_towel();
  put_away_shampoo();
}
```

In this example, the general logic is clear, and the cleanup code is always run, whatever disaster happens. This example includes a catch-all `catch` block at the end. It is a good idea to put one of these in your program's `main()` function so that it can deliver a more polite message than "illegal instruction." But because you will then have no information about what caused the problem, it's a good idea to cover a number of known cases first. Such a catch-all must be the last `catch` block; otherwise, it will mask more specific errors.

It is also possible to use a trick that Perl programmers use: If the `fail()` function returns a `bool`, then the following expression is valid C++ and does exactly what you want:

```
dry_hair() || fail("no towel");
lather() && rinse() || fail("no hands!");
```

If `dry_hair()` returns true, the `or` expression must be true, and there's no need to evaluate the second term. Conversely, if `dry_hair()` returns false, the `fail()` function would be evaluated and the side effect would be to throw an exception. This short-circuiting of Boolean expressions applies also to `&&` and is guaranteed by the C++ standard.

Case Study: A Reverse-Polish Calculator

The first serious scientific calculators made by Hewlett-Packard had an eccentric way of entering expressions backward. You entered the numbers and then applied the operation. It is a classic use of the stack idea: Two numbers are pushed onto the stack, and then a multiply function operates on these numbers and replaces them with the result. Hewlett-Packard's many fans claimed that they could type in expressions faster than people using good old-fashioned *infix notation* because they did not need parentheses. Infix notation puts the operator between the operands, like `10 + 20`. Reverse Polish Notation (RPN—Polish as in the nationality) writes the operator afterward, like `10 20 +`. A surprisingly popular programming language called FORTH (originally developed to point telescopes) used RPN. So we have:

10 20 + 2 * RPN is   (10 + 20) * 2 infix

1.2 sin 1.3 cos + RPN   is   sin(1.2) + cos(1.3) infix

In this case study, you need to write a small program that lets Hewlett-Packard calculator fans do arithmetic. If you give the program a reverse-Polish expression as a command line, it evaluates the expression immediately; otherwise, it assumes that expressions are read from standard input and terminated with a period (`.`), which means "display the value."

Using Stacks

A stack module is defined in `stack.cpp`. If there are already two numbers on the stack, then the stack module will perform the addition, as follows:

```
;> Stack::push(10); Stack::push(20);
;> Stack::push( Stack::pop() + Stack::pop() );
;> Stack::tos();   // top of stack
(double) 30.
```

The following example is a first version of the reverse-Polish calculator RP —or at least the first version that worked.

If you write the operations as in the preceding example—for example, `push(pop() / pop())`—the operands end up in the wrong order. (Try it and see.) Another surprise with this example is that passing `3 5 *` as a command line did not work. When I dumped out the command-line string, it had every file in the directory! By default, as you saw earlier, C++ programs compiled with GCC expand any wildcards, such as `*`. After some scratching around in the runtime library source, I found a way to switch this off.

The actual calculator is contained in the `eval()` function. It is basically a `while` loop that reads in tokens separated by spaces. If the token's first character is a digit, then RP assumes it is a number and converts the string to a number. This number is then pushed onto the stack. If the token is “.”, then it means that the expression is finished and RP returns the top of the stack. (So if you gave it “10 .” it would push 10, pop 10, and then display this value.) Otherwise, if the token is an operator, then RP applies that operation to the top two elements of the stack. Finally, functions like `sin()` and `cos()` are implemented.

EXAMPLE

```
// RP.CPP - Reverse-Polish Calculator vs 1.0
#include <iostream>
#include <stringstream>
#include <string>
#include <cstdlib>
#include <cmath>
#include <cctype>
using namespace std;

#include "stack.h"

double str2float(string s)
{
  return atof(s.c_str());
}

double eval(istream& in)
{
  using namespace Stack;
  string tok;
  while (in >> tok) {
     if (isdigit(tok[0])) push(str2float(tok));
     else if (tok == ".") return pop();
     else {
       double val = pop();
       if (tok == "+") push(pop() + val);
       else if (tok == "-") push(pop() - val);
       else if (tok == "*") push(pop() * val);
       else if (tok == "/") push(pop() / val);
       else if (tok == "sin") push(sin(val));
       else if (tok == "cos") push(cos(val));
     }
  }
  return pop();
}

// this prevents 'globbing' of the command line
// in Mingw32. (The other compilers have it off by default)
// (see comment in init.c from the runtime source)
int _CRT_glob = 0;

int main(int argc, char *argv[])
{
 if (argc > 1) { // command-line arguments
   string exp;
   for(int i = 1; i < argc; i++) { // stitch the arguments together as a string
      exp += ' ';
      exp += argv[i];
   }
   istringstream ins(exp);  // and pass the string to eval() as a stream
   cout << eval(ins) << endl;
 } else {
  for(;;)  // pass standard input to eval()
    cout << eval(cin) << endl;
 }
 return 0;
}
```

To build the reverse-Polish calculator, you need to first build the stack module with `-c`. Then compile `rp.cpp` and link it with `stack.o`, using the following commands:

```
C:\ucw\chap4>c++ -c stack.cpp
C:\ucw\chap4>c++ rp.cpp stack.o
C:\ucw\chap4>a 10 5 /
2
C:\ucw\chap4>a
1.2 sin 1.3 cos + .
1.19954
^C
```

In the preceding example, this program is first run with command-line arguments, which `main()` builds up into a single string by using spaces. Because `eval()` expects an input stream, you use `istringstream` to pass it the resulting string.

If there are no command-line arguments, `eval()` is passed the standard input stream `cin`. But because you have a loop that goes on forever here (because `cin >> tok` will always succeed), you have to stop the program by using Ctrl+C. This is not very elegant. (Fortunately, you can nearly always get yourself out of trouble this way.)

Adding Error-checking to RP

To make this program more solid, you need to pick a character, such as ‘q’ to mean quit, and you need to look out for several error conditions:

- Although it's unlikely that the stack will overflow, it's easy for the stack to *underflow*, which means you are trying to pop an empty stack.
- You need to watch for division by floating-point zero, which is not a runtime error by default.
- You need to watch for domain errors (for example, trying to calculate the square root of a negative number).

Users would also like to define variables. To set pi to 3.1412, you use `3.1412 pi =`, which is consistent with RPN but not convenient. It is tricky because we meet the variable `pi` before we meet `=`. You can use a `map` from `strings` to `doubles` to define variables. That is, you can store the value of `pi` as `value_map["pi"]`. Fortunately, this `map` always has a sensible value, so you push the value of any token that begins with a letter (see the line ending with [4] in the following example), and keep track of both the token and the value that was on the stack. Then when you encounter `=`, these values are available (notice the line ending with [5]).

EXAMPLE

```
double non_zero(double d)
{
 if (d == 0.0) throw string("must not be zero");
 return d;
}

double non_negative(double d)
{
 if (d <= 0.0) throw string("must not be negative");
 return d;
}

double pops()
{
 if (Stack::empty()) throw string("Stack empty!");
 return Stack::pop();
}

map<string,double> value_map;

double eval(istream& in)
{
  using namespace Stack;
  string tok, last_tok;
  double last_val=0,val;
  while (in >> tok) {
     if (isdigit(tok[0])) push(str2float(tok));
```

```
    else if (tok == ".") return pops();   // end of expr
    else if (tok == "q") throw 0;        // finished!  [1]
    else {
      val = pops();
      if (tok == "+") push(pops() + val);
      else if (tok == "-") push(pops() - val);
      else if (tok == "*") push(pops() * val);
      else if (tok == "/") push(pops() / non_zero(val));  //[2]
      else if (tok == "sin") push(sin(val));
      else if (tok == "cos") push(cos(val));
      else if (tok == "sqrt")
           push(sqrt(non_negative(val)));                //[3]
      else if (tok == "dup")  { push(val); push(val); }
      else if (tok == "=")
          value_map[last_tok] = last_val;                //[4]
      else if (isalpha(tok[0])) {
        push(value_map[tok]);                            //[5]
        last_tok = tok;
        last_val = val;
      }
    }
  }
  return pop();  // ran out of input...
}
```

The three value-checking functions are interesting (see the lines that end with [2] and [3]). It is difficult for a function like `non_zero()` to return a value that means “bad value,” since what floating-point value could you choose to mean an error? So `non_zero()`, `non_negative()`, and `pops()` all throw an appropriate exception. Wherever you would use `Stack::pop()`, you instead use `pops()`. I may now say `sqrt(non_negative(val))`, and `sqrt()` will never be called with a negative argument. Instead, an exception will always be thrown. The point is that error-checking need not involve lots of ugly `if` statements.

The function `eval()` also needs to indicate that the user is tired and has terminated the session by using `q`, so you throw another kind of exception (an `int`), which guarantees that normal termination is different from an error. You then call `eval()` from an error-checked version function called `safe_eval()`:

```
double safe_eval(istream& in)
{
 try {
   return eval(in);
 } catch(string err) {
   cerr << "error: " << err << endl;
```

```
    return 0.0;
  }
}
```

Notice that `safe_eval()` does not catch the `int` that `eval()` throws if the user types `(q)`. This means that the caller of `safe_eval()` can catch the `int` exception and bail out of the loop, as in the following code:

```
try {
 for(;;)
   cout << safe_eval(cin) << endl;
} catch(int ierr)  {
  // jump out of loop!
}
```

Some people do not think it's a good idea to use exceptions for normal termination of a program. They argue that exceptions must be at least unusual, if not actually errors. But this example demonstrates how using different kinds of exceptions can enable you to specify precisely where you want an error to be handled. The function `safe_eval()` is only interested in trapping expression errors (and telling the user about them); it passes through the quit message that `eval()` sends to the `main()` function.

You can try making several improvements to the reverse-Polish program. For example, you can handle trigonometry with degrees, not radians. It would be useful to save any defined variables to a file (remember that you can iterate over a map). Even if doing arithmetic backward isn't your cup of tea, the idea of a stack machine is very common in computer science. On Intel platforms, the floating-point machine instructions are precisely comprised of `push`, `pop`, and operations that act on a stack.

What's Next

In this chapter you applied your knowledge of functions to building full C++ programs, which can be compiled as stand-alone executables. In the next chapter you will learn about structures and pointers; structures are a powerful way of 'packaging' complex data together. Pointers allow you to generate very dynamic data structures.

5

Structures and Pointers

You have learned to structure programs by using functions and to structure data by using arrays and containers. Functions, arrays, containers, and program input/output were all that FORTRAN and BASIC gave programmers. Modern languages, including C++, give you the ability to group different kinds of data together and create dynamic references.

In this chapter you will learn

- How to declare and use C++ structs
- Why C++ structs are a superior way to organize data
- How to manipulate data via pointers

User-Defined Structures

A structure or `struct` is a user-defined collection of data. It is superior to using arrays because each item can have a different type and a different name. In the following section, you will see the advantages of using user-defined structures.

Arrays Are Not Enough

Data often appears in groups, or *aggregates*. For example, in the case study in Chapter 3, "Arrays and Algorithms," which performs a simple stock price analysis, there are five numbers for each trading day: the four prices and the trading volume. Similarly, measurements from a meteorological station would include temperatures, air pressure, wind speed and direction, and so on. In these cases, all the numbers could be floating-point numbers, so an array could be used. Then `mm[k][2]` would represent, say, the third number of the `k`th measurement; however, this becomes very confusing. There are other disadvantages; you cannot copy arrays with simple assignment, for instance.

Now consider a case in which a data aggregate consists of different kinds of information. To characterize a person, you need two names, an ID number, a phone number (or two), and an address; the data types are mixed (for example, strings, integers). The C++ `struct` was designed to represent this kind of data.

Defining New Types

A `struct` (short for "structure") definition creates a new type, and it does not reserve space for a variable. The new type defines how memory is to be organized, by declaring *member variables* or *fields*. After the new type is defined, you can use it to declare variables like any other C++ type. For example, consider a point on a graph, which has two coordinates `x` and `y`. The following code defines a two-dimensional point type `P` as a `struct`. Until a variable `p` of type `P` is declared, no memory is reserved; the variable is laid out as shown in Figure 5.1, with the member `x` at zero offset and the member `y` at an offset of 4 bytes (which is the size of an `int`.) To access the members, the member selection (or dot) operator (`.`) is used.

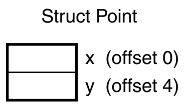


Figure 5.1: *Memory layout of a `struct Point`, showing the offsets of the two fields.*

EXAMPLE

```
struct Point {
  int x;
  int y;
};
;> Point p;
;> p.x = 10; p.y = 20;
(int) 10
(int) 20
;> Point q = p;
;> q.x + q.y;
(int) 30
```

Notice that you can initialize a `struct` variable by using another variable of the same type (`Point q = p`.) Similarly, you can assign such variables to each other. If you represented points as arrays of two `int`s, then you wouldn't be able to copy them as easily as in the preceding example. You can use expressions such as `q.x` anywhere you would use a normal C++ variable reference.

The following is an example of a structure that describes a person:

```
struct Date {
  short year;
  unsigned char month,day;
};
struct Person {
   string first_name;
   string last_name;
   Date birthday;
   long id;
   string postal_address;
   string email_address;
   long phone_no;
};
;> Person p;
;> p.birthday.year = 1965;
```

Notice that structures can contain just about any type, including other structures. `Person` contains a member `birthday`, which is a `Date`. In cases like this, you use the dot operator as many times as necessary (as in, `p.birthday.year = 1965`.) You can initialize structures by using a set of values within braces. UnderC does not currently support this, but with standard C++ you can initialize a `Person` structure as follows:

```
Person fred = {"Fred","Jones",{1965,03,02},2343222,"",
                "fjones@cplusplus.com",4552334};
```

As with the enum and struct statements, the last brace in a struct is followed by a semicolon.

Passing structs to Functions

struct variables can be used like any other variable type. If p1 and p2 are both of type Person, then p1 = p2 will copy all the members of p2 to p1. struct variables can be passed as arguments to functions; by default they are passed by value. The function int getx(Point p) { return p.x; } returns the x coordinate of a Point p. The function call getx(p1) will copy the actual argument (p1) into the formal argument (p). It is also possible to pass a struct by reference. Such arguments can be very convenient ways to pass a lot of information to a function. Functions may also return struct values.

Using Structures Instead of Many Parameters

Functions that have too many arguments are not only difficult to type but easy to get wrong. For example, read_any_stock() (discussed in Chapter 3, "Arrays and Algorithms") has five arguments, and it's easy to mix up the prices. The following function compares dates; it is similar to int compare_dates(string d1, string d2) as discussed in that case study, so I have not written it out fully:

```
int compare_dates(int y1, int m1, int d1,
                  int y2, int m2, int d2)
{
  int cmp = y2 - y1;
  if (cmp != 0) return cmp;
  ...
}
```

When I call such a function, I easily get confused about the order of the arguments. I know the function needs two dates, but is the day before the month (as I usually write dates)? The Date struct makes this kind of function easier to use, especially if you write a special function date() for making Date objects. Look at the next definition of compare_dates():

```
Date date(int y, int m, int d) {
  Date dd;
  dd.year = y;  dd.month = m;  dd.day = d;
  return dd;
}
int compare_dates(Date d1, Date d2)
{
  int cmp = d2.year - d1.year;
  ....
}
;> compare_dates(date(2001,07,17),date(2001,06,05));
(int) -1
```

The function now has only two arguments; note that the function `date()` and type `Date` have distinct names.

NOTE

You may have noticed that I write `structs` and `typedef` names with an initial capital letter. This is part of the naming convention I like to use; any type names are `WrittenLikeThis`, and any variables or functions are `written_like_this`. The only exceptions are the standard library types (such as `std::list`) and when linking to someone else's functions, such as the Windows API (e.g. `GetWindowText()`).

Using the `Date` struct has some advantages over using a string representation. First, it is more compact. The struct can be packed into 4 bytes (two bytes for the year and one each for month and for day). Second, it is much more efficient in terms of speed and code size. Finally, it is automatically in the right format (year,month,day) unlike a date string. A person may easily pass "20 June 2001" to a function expecting "2001-06-20". A `Date` variable may still may be invalid, of course (e.g. month is greater than 12).

Passing Structures by Value

Remember from Chapter 3, "Arrays and Algorithms," that `vectors` are always passed by value. The same applies to user-defined structures. This means two things: You don't have to worry about accidentally modifying a passed structure, and you do have to sometimes worry about the cost of all that copying. A function can also return a structure, but it isn't a good idea for large structures such as `Person`, which is introduced earlier in this chapter. For compact structures such as `Point`, you can use this fact to make a function that constructs points, as in the following example:

EXAMPLE

```
void show_point(Point p) {
 cout << p.x << ' ' << p.y << endl;
}
Point make_point(int a,int b) {
  Point p;
  p.x = a;  p.y = b;
  return p;
}
;> Point pp = make_point(2,3);
;> show_point(pp);
2 3
;> make_point(5,7).x;
(int) 5
```

Note that you can follow any expression (such as a call to `make_point()`) that returns a structure with the dot operator.

Reference Types

In the section "Containers," of Chapter 3, I mentioned that you can force an argument to be passed by reference by using the `address-of` operator (`&`) in the argument type. This operator means that the argument has a *reference type*. Reference types can be declared anywhere, but they must be initialized with a proper variable, as in the following example:

```
;> int& ri;
CON 3:uninitialized reference
;> int k = 1;
;> int& kr = k;
;> kr;
(int&) 1
;> kr = 2;
(int&) 2
;> k;
(int) k = 2
```

The reference `kr` is, in effect, just another name for `k`; whatever you do to `kr` will affect `k` and vice versa. This is because they occupy the same address in memory: `kr` is an *alias* for `k` (that is, another name for the same thing). References are often used when a function has to pass more than one item of information back to the caller. The function `modify_ij()` is intended to return two integer values; by passing `i` and `j` as references, they can both be modified:

```
bool modify_ij (int& i, int& j) {
   i = 0; j = 0;
   return true;
 }
; int m = 9;
;> modify_ij(k,m);
(bool) true
;> k; m;
(int) k = 0
(int) m = 9
```

A function may return a reference, but there can be problems. If you are returning a reference to a local variable, then by the time the function has returned to its caller, that local variable may no longer exist. For instance, if you declared the return type of `make_point()` as `Point&`, you might get very strange results. Returning references may work on one machine, but not on another.

`const` References

Passing a `struct` variable as a reference is usually fastest for a nontrivial structure, but this allows the structure to be modified by the function. You use the `const` qualifier to insist that an argument be passed as a reference to a constant object, as in the following example:

```
;> int var = 1;
;> const int& var_ref = var;
;> var_ref = 2;  // error! Can't modify a const reference!
CON 23:Can't assign to a const type
;> string fullname(const Person& p) { return p.first_name + " " + p.last_name; }
```

You should declare the function `compare_date(const Date& d1, const Date& d2)` rather than `compare_date(Date d1, Date d2)`. In the case of `compare_date()` there is probably little difference; `Date` would be packed into a 4-byte word (but it might end up as three words if the compiler decides to use at least one word per field—this is called *word alignment*). Large structures such as `Person` would be expensive to pass by value. You should qualify a reference with `const`, unless you intend to modify the value. Experienced programmers assume that an object will be modified if they don't see `const`.

Arrays of Structures

You can define arrays of user-defined types, and you can define standard containers involving them. You will often need to keep collections of objects like `Person`s or `Point`s.

Plain Arrays

You declare arrays of structures the same way you declare any other array, and you access them by using subscripting, as in the following example:

```
;> Point pts[5];
;> sizeof(pts);
(const int) 40
;> sizeof(Point);
(const int) 8
;> sizeof(pts)/sizeof(pts[0]);
(int) 5
;> pts[0].x = 1;
(int) 1
;> pts[1] = make_point(30,40);
(Point&) Point {}
;> show_point(pts[1]);
30 40
```

You can initialize arrays of structures. UnderC does not currently support such initialization, but the following is a small program that compiles with standard C++, together with its output:

EXAMPLE

```
#include <iostream>
using namespace std;

struct Point {
  int x,y;
};

Point pts[] = {
   {234,543}, {260,677}, {302,700}, {244,760}
 };

int main()
{
  for(int i = 0; i < sizeof(pts)/sizeof(Point); i++)
    cout << pts[i].x << ' ' << pts[i].y << endl;
  return 0;
}
```

OUTPUT

```
234 543
260 677
302 700
244 760
```

In this example, be careful about the semicolon after the brace in the list of initial values; the error message is obscure if you leave out the semicolon!

Lists and Vectors of Structures

Often a `vector` of structures is more appropriate than an array, if the number of elements is not known at compile time. Otherwise, `vector`s of structures and arrays of structures are used in the same way. Similarly, `list`s of structures can be very useful, as in the following example:

```
;> vector<Point> vp;   // initially empty
;> vp.push_back(make_point(10,20));
;> vp[0].x;
(int) 10
;> list<Point> lp;
;> for(int I=0;I<10;I++) lp.push_back(make_point(I,I*I));
;> lp.back().x;
(int) 9
```

When you have any container of structures, it is straightforward to iterate over that collection:

```
void dump_points(const list<Point>& lp) {
  list<Point>::iterator lpi;
  for(lpi = lp.begin(); lpi != lp.end(); ++lpi)
    cout << '(' << (*lpi).x << ',' << (*lpi).y << ")\n";
}
```

It is very common to find expressions such as `(*lpi).x` when using iterators; because the dot operator (`.`) has a greater precedence than the dereference operator (`*`), you need to include parentheses in the preceding expression (`*lpi.x` is read as `*(lpi.x)`, which is wrong.) It is such a common operation that an operator shortcut, `lpi->x`, exists. As mentioned in the "Iterators," section of Chapter 3 a map has a `find()` method, which returns an iterator, but I didn't tell you how you can use that iterator (apart from saying that it may be equal to `end()`). The iterator refers to a pair of values: the key (which we know) called `first` and the value (which we are trying to find out), called `second`. Rather than using `(*imsi).first`, you can use `imsi->first`. In this example I have declared a `map<string,int>` and associated the string "fred" with the number 648. The iterator returned from `find()` has information about both the key and the value:

```
;> map<string,int> msi;
;> msi["fred"] = 648;
;> map<string,int>::iterator imsi = msi.find("fred");
;> cout << "key = " << imsi->first
        << " value = " << imsi->second << endl;
key = fred value = 648
```

Pointers

Pointers have long been a source of trouble for beginners and experts alike. It takes a while for beginners to get used to pointers, and they are a fertile source of bugs in even professionals' programs. The pointer is an animal that is never entirely tamed, and it should be treated with respect. Pointers are no longer as crucial in basic C++ programming as they were in C, which is why I've delayed talking about them until this chapter. Indeed, Java deliberately eliminated pointers because they are potentially unsafe, although they are still there, disguised as references.

Pointers as References

You can think of memory as a large array of bytes, up to 4GB in length on a 32-bit machine. The address of a variable is the index into that array. You can define a *pointer value* as being an address. To find the address of a variable, you use the address-of operator (`&`), which we previously saw used in reference declarations (in "Reference Types"). Pointer values are traditionally expressed in hexadecimal (that is, base 16, not base 10), but you can typecast a pointer into an integer if you want to see the value in decimal, as in the following example, which also shows the addresses of two adjacent variables `i1` and `i2`:

```
;> &i;
(int*) 72CD30
;> (int) &i;
(int&) 7523632
;> int i1=1,i2=2;
;> &i1; &i2;
(int*) 72CD54
(int*) 72CD58
```

Notice in this example that the address of `i2` is 4 bytes ahead of the address of `i1`. But don't depend on facts like this! These pointer values will not be the same on your machine. Generally, you should not use the actual values of these addresses. They will change as a program changes and will probably be different if the program is rebuilt on another platform.

A pointer is a variable that contains a pointer value. That is, it is a reserved area of memory, 4 bytes in size on most machines, that contains the address of some other variable. You declare and access pointers by using the dereference operator (`*`). The following example shows how. Figure 5.2 shows how `i1`, `i2`, and `pi` are laid out in memory for this particular example.

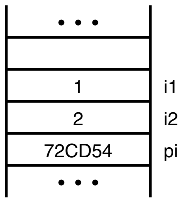


Figure 5.2: *The pointer variable `pi` contains the address of the integer variable `i1`, which contains `1`.*

EXAMPLE

```
;> int i1 = 1, i2 = 2;
;> int *pi = &i1;
;> *pi;
(int) 1
;> *pi = 10;
(int) 10
;> int& ri = i1;
;> int *p2 = pi;
;> *p2;
(int) 10
```

The result of *dereferencing* a pointer (*pi, in this case) is the value contained in the original variable i1. *pi is written exactly as iterator expressions such as *lpi (see "Iterators" in Chapter 3); a pointer is a special kind of iterator. *pi acts as an alias for i1, in the same way that the reference ri is an alias for i1. Both *pi and i1 refer to the same memory. You can have as many pointers as you like referring to the original variable. References are in fact a restricted kind of pointer, and they automatically dereference themselves (that is, you don't need *).

CAUTION

More than one pointer can be declared at once, but each new pointer must be preceded by *, as in the following example:

```
;> int *p3, *p4, p5;
;> p5 = &i1;
CON 52: cannot convert from 'int*' to 'int'
```

This code declares p3 and p4 to have type int * but declares p5 to have type int. p5 is just a plain integer, and trying to put a pointer value into it results in an error. If you are used to a more relaxed language, C++'s insistence on types matching can become irritating. However, it prevents you from writing foolish or badly behaved code.

You can use any C++ type as the base type of a pointer. It is important to note that C++ will not let you automatically convert one pointer type to another. If you force the conversion by using a typecast, you are likely to get garbage. This is similar to what happens with passing arrays to functions; a double variable is 8 bytes and is organized very differently from a 4-byte integer. There is no translation involved with pointer assignments, such as when a double value is translated to an integer value. In the following example, a pointer to int is copied into a pointer to double, and the result of dereferencing that pointer is displayed. The bit pattern representing the integer 10 is some arbitrary (and very small) floating-point number.

```
;> int i = 10;
;> int *pi = &i;
;> double *pd;
;> pd = pi;   // not allowed!
CON 55: cannot convert from 'int*' to 'double*'
;> pd = (double *) pi; // force it!
(double*) 72CD58
;> *pd;
(double&) 1.67896e-306
```

Using pointers with structures is probably the most common use of pointers. Note that just as with iterators, the clumsy (*ppt).x can be written as ppt->x, as in the following example:

```
;> Point pt = make_point(500,200);
;> Point *ppt = &pt;
;> (*ppt).x;
(int) 500
;> ppt->x;
(int) 500
```

Some operating system functions use structures. For instance, the standard C header `time.h` contains the `tm struct`, which includes all you need to know about the time, and allows the time functions to pass back a large amount of information, as in the following example:

EXAMPLE

```
struct tm {
        int tm_sec;     /* seconds after the minute - [0,59] */
        int tm_min;     /* minutes after the hour - [0,59] */
        int tm_hour;    /* hours since midnight - [0,23] */
        int tm_mday;    /* day of the month - [1,31] */
        int tm_mon;     /* months since January - [0,11] */
        int tm_year;    /* years since 1900 */
        int tm_wday;    /* days since Sunday - [0,6] */
        int tm_yday;    /* days since January 1 - [0,365] */
        int tm_isdst;   /* daylight savings time flag */
  };
```

The following function `today()` uses `tm` to generate a `Date` structure for today's date:

```
Date today() {
  time_t t;
  time(&t);
  tm *ts = localtime(&t);
  Date dd;
  dd.year = ts->tm_year + 1900;  // years since 1900...
  dd.month = ts->tm_mon + 1;     // it goes 0-11...
  dd.day = ts->tm_mday;
  return dd;
}
```

In this example, the type `time_t` is a `typedef` for a `long` integer; the `time()` function returns the number of seconds since midnight January 1, 1970 (which is when UNIX started counting time). `localtime()` returns a pointer to a `tm` struct, which you can transfer to the `Date` structure. The time functions in this example are a little eccentric, and there are things to watch out for (for example, the month field goes from 0 to 11). It is a good idea to keep this kind of unexpected behaviour wrapped up safely in a function so that you do not have to keep remembering to add 1900 to `tm_year`. One advantage of using a `struct` to return information from a function is that you need refer to only what you need, and you can ignore the rest.

Structures can themselves contain pointers. For instance, here is an `Account` structure that keeps information about money outstanding:

```
Struct Account {
  Double amount;
  Person *debtor;
  Date when_due;
 };
```

Why would you keep a pointer to a structure? Consider that any given `Person` object probably has a number of accounts (`Account` object), and `Person` is a rather large `struct`. So keeping one `Person` object shared among several `Account` objects would save memory and disk space. But the most important reason is that you keep one (and only one) `Person` object and thus there is no redundant information floating around. Sooner or later, the person's contact details change, and if you didn't share one object, you would have to go through every structure in the system that contains that `Person` object and update the details. (Some government departments seem to work like this.)

Pointers can also be used to pass variables by reference:

```
;> void update_value(int *p) { *p = 0; }
;> update_value(&i1);
;> i1;
(int) i1 = 0
```

C, which does not have reference variables, uses pointers to pass structures by reference. Notice that you have to explicitly extract the address of the argument you want to pass by reference, using `&`. Many C++ programmers prefer this style because it shows that the argument of `update_value()` is going to be modified. With references, there is no way you can tell just by looking at the function call whether the argument is going to be modified or not.

What Pointers Refer To

The big difference between references and pointers (apart from needing to say `*p` to extract values from pointers) is that each reference is bound to one (and only one) variable. Pointers can be made to point to new variables, and they can also be made to point to anything, which makes them powerful and also potentially full of problems. Consider the following example. You can put an arbitrary number `648` into a pointer, although you will need an `(int *)` typecast to remove the error message. But usually any attempt to dereference such a bogus pointer results in a *access violation* message. In other words, `648` is considered an illegal address. Because local variables are not initialized automatically, you also need to watch out for uninitialized pointers.

```
;> int pi = 648;
CON 42: cannot convert from 'const int' to 'int*'
;> pi = (int *)648;
(int*) 288
;> *pi;
access violation <temp> (1)
```

By now I hope you're getting worried every time you see a typecast. Sometimes they are necessary, but too many typecasts usually spells trouble. You might wonder why this arbitrary value `648` was considered a bad pointer. Surely if you have 64MB of system memory, you can address everything from 0 to about 64×1024×1024 = 67,108,864, right? But modern operating systems don't work like that; each program (or *process*) gets its own 4GB address space, and only isolated sections of that total range are actually valid memory. System memory is virtual, with chunks of memory (called *pages*) being swapped to and from disk continuously. So you actually have about 100MB of virtual memory available if you have 64MB of random access memory (RAM), but it's much slower to access the disk than to access RAM. As you load more and more programs, your system grinds to a halt because it is constantly trying to page to disk. The separate address space for each process means that a program can't destroy another program's data; the pointer value 0x77DE230 (say) will have different meanings for different programs.

In Windows 9x, the upper 2GB is shared among all programs and is used for dynamic link libraries and the operating system. So your program can still overwrite system memory. This is a major reason Windows NT is more reliable than Windows 9x

In C++, you can have pointers to pointers. Remember that a pointer is basically a variable, so it has an address or pointer value, which you can store in another variable of a suitable type. Again, the style of declaration is meant to resemble how the pointer will be used; to dereference `int** ppi` (read as "pointer to pointer to `int`"), you would use `*ppi`. In the following code, I have `ppi`, which points to `pi`, which points to `i1`. `*ppi` will be the value of the pointer `pi` (that is, the address of `i1`). `**ppi` will be the value of `i1`.

```
;> int *pi = &i1;
;> int **ppi = &pi;
;> *ppi;
(int*) 72CD54
;> **ppi;
(int) 10
;> void *pv = &i1;
;> double f;
;> pv = &f;    // (double *) will convert directly to (void *)
(void*) 72CE2C
;> *pv;  // this will be an error!
```

There are two things to remember about `void` pointers: Any pointer will happily convert to a `void` pointer, and you cannot dereference a `void` pointer because the system does not know what the pointer's base type is.

The `NULL` Pointer

The `NULL` pointer is usually simply defined to be 0, which is an exception among integers; it matches any pointer type without typecasting. Trying to access memory at logical address zero always results in an access violation on machines that have memory protection. UnderC recognizes this as a special problem, but most compilers do not distinguish this case in their runtime errors. With the GNU compiler, a program will say something like "A has caused an error in A.EXE. A will now close" and give some helpful advice about restarting your computer if problems persist. (This is ultimately why I went to the trouble of creating UnderC: I spent years being frustrated with weird runtime errors and slow compilers!) Here you see UnderC's more friendly runtime error:

```
;> pi = NULL;
(int*) 0
;> *pi;
NULL pointer <temp> (1)
```

Java programmers are likely to regard this as yet more proof that C++ is not to be trusted. With C++, you should always check to see whether a pointer is `NULL` because many programs use this value to indicate a special state.

Writing Structures to Binary Files

Up to now you have only written plain ASCII text to files. You can inspect such files with a text editor such as Notepad, and they contain no unprintable characters. The main difference between binary files and ASCII text files is that binary files can use all 256 ASCII characters, whereas text files tend to keep to a subset of the lower 127 characters.

The other difference between these types of files has to do with human stubbornness; when Microsoft developed DOS, it decided that the ends of lines of text should be indicated with two characters, a carriage return (`cr`) and a linefeed (`lf`). The UNIX people just used `newline`, which is written like `\n` in C character strings. (Mac people just use carriage return, so everyone is different.) When you write the string `"this is a line\n"` to a text file, `"\n"` must be translated into `"\r\n"`; the opposite translatation must occur when you read in a line.

Accessing a file in binary mode involves no filtering or translation. Streams default to text mode, so a special constant is needed when you open a file using `ofstream`. Here is a small function that writes a few integers directly to a file:

```
void binarr(int arr[], int n) {
  ofstream out("arr.bin",ios::binary);
  out.write((char *) arr, n*sizeof(int));
}
;> int a[] = {1,2,3,4};
;> binarr(a,4);
```

You use another of the `ios` constants (`ios::binary`) to indicate that any character written out to this file should not be translated. An integer might include a byte with the value of `\n` (that is, hex `0x0A` or decimal `10`), and you don't want `write()` inserting an extra `\r`. UNIX programmers should particularly be aware of this potential problem with Windows.

The `write()` method takes two arguments: a pointer to `char` and the total size (in bytes) of the data to be written. Usually, you have to put in a typecast to force the type to be `(char *)`; if it were `(void *)`, then any pointer would freely convert. If you now look at the created `arr.bin` file, you'll see that it is exactly 16 bytes (4 integers of 4 bytes each.) You can use a utility such as DOS `DEBUG` to look at the first few bytes of the file:

```
c:\ucw\examples\chap5> debug arr.bin
-d 0100,010f
1AC2:0100  01 00 00 00 02 00 00 00-03 00 00 00 04 00 00 00
-q
c:\ucw\examples\chap5
```

Sure enough, `1`, `2`, `3`, and `4` each take up 4 bytes. Notice that the least significant part of each number (that is, the *low byte*) appears first. Because the most significant part is at the end, this is called *big-endian*, which is how Intel processors arrange bytes. (*Little-endian*, on the other hand, is how Motorola processors arrange the bytes, with the least significant part at the end; you use little-endian when you write down numbers: 45,233.)

I mention big-endian and little-endian now to raise a warning: Binary files are not fully portable across different platforms. Consider someone trying to read the four integers in `arr.bin` on a non-Intel machine. A Macintosh would be confused by the byte ordering, because it has a little-endian processor. A Silicon Graphics workstation is a 64-bit machine (making `sizeof(int)` equal to 8), so the integers in the file would be the wrong size. In recent years, therefore, there has been a move toward ASCII data standards, such as Hypertext Markup Language and rich text format.

So what are the advantages of binary files?

- They are usually more compact and are definitely more efficient than ASCII files. The four integers in the last example were transferred without any translation. To write out four integers as text, they must be converted into decimal representation.
- They are more secure than text files. Users strongly believe that any ASCII files are for them to edit, and they will then blame you when the application no longer works.
- They are particularly good at accurate storage of floating-point numbers.
- It is easy to write a structure directly into a binary file (as we will see.)
- Most importantly, you have true random access to the file. That is, you can move to any arbitrary point in the file, given the offset in bytes (this is often called 'seeking') The text translation involved in text files makes such access unreliable.

NOTE

There are some things you cannot write to disk very easily. It is meaningless to write a pointer out directly because it will not have any meaning if you read it in again. A pointer refers to a specific address, which is kindly (and temporarily) allocated to you by the operating system. Besides, copying the pointer does not copy the data! Not only pointers are affected by this restriction; if you look under the hood, you will see that standard `strings` are basically structures that contain pointers. So structures such as `Person`, which contains `strings`, are unsuitable for direct binary input/output.

Allocating Memory with `new` and `delete`

So far you've used two methods for allocating memory to variables. The first method is through a global or static declaration, where the compiler reserves space when it is generating the code. (Note that static allocation in UnderC is limited to about 1MB.) The second method is by using local variables (called automatic allocation), which has the advantage that the memory is used only within the function call. The third kind of memory allocation is dynamic: At any point, a program asks for a block of memory from the *heap*, which is the pool of free system memory. The heap is only limited by the amount of memory available on your system. This allocation is achieved by using the `new` operator:

```
;> int *p = new int;
;> *p = 2;
(int) 2
```

Earlier in this chapter, we spoke of pointers referring to other variables; that is, a pointer is another way of accessing a variable. Some people speak of dynamically allocated pointers as being *nameless variables*, which are accessible only through the pointer. Figure 5.3 shows this situation; the pointer value is outside the block of statically allocated memory.

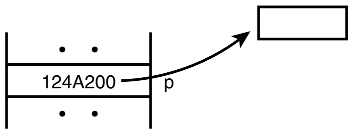


Figure 5.3: *The pointer* `p` *points outside the usual variable memory.*

A marvelous thing about dynamic memory is that you can give it back to the system by using `delete`. When you are finished with a pointer, you should give the block of memory back. After you delete the pointer, that memory no longer belongs to you, and the pointer is called a *dangling pointer*. It is like a phone number after the service has been disconnected. By deleting a block, you are giving permission to the system to do anything it likes with the block, including giving it to some other part of the program. Accessing dangling pointers can cause problems that are particularly hard to trap. Here is how you use `delete` to give memory back to the system:

```
;> int *p = new int;   // ask for a 4 byte block
;> *p = 2;
(int) 2
;> delete p;   // give the block back
;> *p;        // p is no longer valid
(int) 9773312
```

It is useful to write functions that create pointers to objects and guarantee that the resulting object is properly initialized. This function guarantees that the name fields of a `Person` are properly set:

```
Person *new_person(string fname, string lname)
{
  Person *pp = new Person;
  pp->first_name = fname;
  pp->last_name = lname;
  return pp;
}
```

It is possible for a struct to contain a pointer to an object of the same type. Obviously, you can't have a member that is the same `struct` (because the system doesn't know how big the `struct` is until it's defined), but a member can be another pointer because all pointers are the same size.

Imagine that the struct Person discussed earlier in this chapter has an extra field, parent, of type Person *. You can set up a chain of Person objects (as long as parent is initialized to NULL):

EXAMPLE

```
;> Person *fred = new_person("Fred","Jones");
;> fred->parent = new_person("Jim","Jones");
;> fred->parent->parent = new_person("Joshua","Jones");
;> for(Person *p = fred; p != NULL; p = p->parent)
     cout << p->first_name << endl;
Fred
Jim
Joshua
```

The parent field being NULL at the end is an example of a NULL pointer that is used to indicate some condition—in this case, the end of Fred's ancestors. The for loop may seem unfamiliar, but remember that the second section (p != NULL) is tested before the output statement, and the third section (p = p->parent) executes after the output statement, moving you to the next parent in the chain. This curious arrangement is called a *linked list*, and in fact it is how the standard list is implemented. Figure 5.4 shows how these Person objects are linked to each other (with the name fields indicated as pointers to emphasize that the names are not stored inside the struct). But you would rarely have to write such code since std::list already implements a linked list. It would in fact be better to use list<Person> in this case.

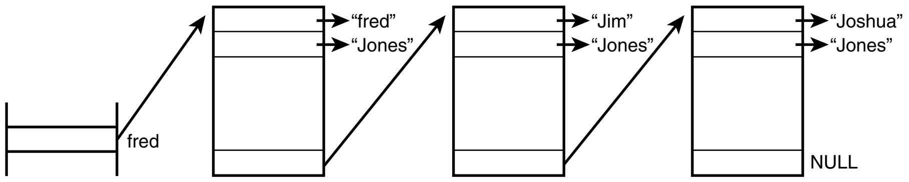


Figure 5.4: *The pointer fred points to a chain, or list, of Person objects.*

NOTE

You may be wondering why pointers are so important, given that they seem to be a major cause of program bugs. The primary use of pointers is to access dynamic blocks of memory. You have been using them for this purpose without realizing it, because the standard containers do this all the time. Modern C++ practice prefers to keep pointers hidden within trusted library code, precisely because they can be such trouble. It is best to wrap up a pointer tangle in an easily used package.

Case Study: The Bug Tracking Program Revisited

It is often necessary to revisit an old program and consider how you can improve it. However, this can be a frustrating exercise because the new specifications often can't be easily implemented by using the old scheme, and in such cases a redesign and rewrite is the only viable choice. There is too much bad code in the world already, kept beyond its natural life like some Frankenstein monster.

The first version of the bug-tracking program you created in Chapter 2, "Functions and Control Statements," works, but users are frustrated: They need to track the people reporting and working on the bugs. They would prefer a number of small utility programs that they can run from the command prompt. Also, the date needs to be logged when a bug has been fixed. To implement these changes, you would just need to add extra fields. But the users also want to edit information in the bugs database—for example, they want to be able to add a comment or decide that a particular must-fix bug can be downgraded to a fix-whenever bug. This is not easy to do with a text file because you would have to rewrite the file for each tiny change. You decide to see how a binary file solution would work. This means the old database will have to be converted, but that would have been necessary anyway. It also raises another question: Can future changes be added without breaking compatibility?

Binary Files: Both Readable and Writable

Consider how the previous bug-tracking system remembered what the last `ID` was: It would read the old value from a small text file, increment it, and write it back. Here is one way to do this with binary files (note how you combine more than one `ios` constant to make the file readable and writable):

EXAMPLE

```
int next_id()
{
  int val;
  ifstream in(IDFILE, ios::binary | ios::in | ios::out);
  in.read((char *)&val,sizeof(int));
  ++val;
  // rewind to beginning of file
  in.seekg(0);
  in.write((char *)&val,sizeof(int));
  return val;
}
```

The significance of this for bug reports is that you can move around a binary file and overwrite existing data. So you need to define a suitable structure that describes a bug report. Then to move to the nth record, you need to seek (n-1)*sizeof(S) bytes, where S is the struct. If S is 100 bytes in size, then the offsets are 0, 100, 200, and so on.

What About the Users?

Each bug report has two associated users: the one that reported the bug and the one that signed off the bug as fixed. There are two corresponding dates—the reported date and the fixed date—which are easy to represent with the Date struct. But how do you represent users? Each user has an eight-character username (for example, SDONOVAN), but it seems a waste to copy the whole name into the structure. Besides, what if the username changes and is no longer eight characters? One solution is to store a user ID, which is a simple integer from 1 to the number of users. To translate from a username to a user ID, you use a map, and to translate from the ID to the name, you use an array. But the rest of the program doesn't need to know that. All the clients of the module need to know is that Users::name() translates IDs to names, and Users::id() translates names to IDs; if you later change the representation (say user IDs are no longer consecutive), the rest of the program is not broken. This interface is contained in users.h and looks like the following:

EXAMPLE

```
#include <string>
#include <map>
using namespace std;

#include "users.h"
namespace { // private to this module…
 const int MAX_USERS = 20;
 typedef map<string,int> UserNameMap;
 UserNameMap users;
 string user_ids[MAX_USERS];
}

namespace Users {

// ids start at 1, not 0!
string name(int id) { return user_ids[id-1];  }
int id(string s)    { return users[s]; }
string myself()     { return getenv("USERNAME");  }

void add(string s, int id)
{
  users[s] = id;
  user_ids[id-1] = s;
}
void init()
{
  add("GSMITH",1);
  add("FJONES",2);
  add("SREDDY",3);
  add("PNKOMO",4);
  add("SDONOVAN",5);
}

} // namespace users

// users.h
// Interface to the users module
typedef int User;
const int MAX_USERS = 20;

namespace User {
 string name(int id);
 int    id(string s);
 string myself();
 void   add(string s, int id);
 void   init();
}
```

This approach to storing users avoids the problem of how to write strings to a file, but you can't get away from storing the actual bug description.

Writing Strings

The following is a structure that contains the required information:

```
const int MAX_DESCRIPTION = 80;

struct Report {
   int id,level;
   Date date_reported, date_fixed;
   User who_reported, who_fixed;
   long extra1;
   long extra2;
   int  descript_size;
   char description[MAX_DESCRIPTION];
};
```

In this example, everything is straightforward (`User` is just an integer) except the description, which is a fixed array of characters. There are also two extra fields, which allow some room for expansion. The problem with this example is that there is a limited amount of space for the description.

80 characters is probably enough for most bugs, but occasionally bugs need more description than that.

Let's first handle the issue of saving strings as character arrays. Generally, if you are going to do anything nontrivial to a structure and you want to keep your options open, it is a good idea to write functions that operate on the structure. You can wrap up the dark secrets of how strings are stored in two functions, which you can improve later. These are shown in the following code.

```
string get_description(const Report& report)
{
  return report.description;
}

void set_description(Report& report, string descript)
{
 report.descript_size = descript.size();
 strncpy(report.description, descript.c_str(), MAX_DESCRIPTION);
}
```

The function `get_description()` seems fairly trivial, but quite a bit goes on under the hood. The fixed-length character array is implicitly converted into a variable-length string. `set_description()` requires a useful function from the C runtime library. `strncpy(dest,src,n)` copies at most `n` characters from `src` to `dest` (note the direction). The string method `c_str()` provides the raw character data of the string, which is then copied into the report description field. If there are too many characters in the string, the description is truncated. (As usual, it is very important not to run over the end of the array.) The copy operation should be familiar to a C programmer, but this operation is usually conveniently (and safely) handled by C++ strings. (For more information on C-style string handling, see "The C String Functions" in Appendix B, "A Short Library Reference.")

Adding a bug report is straightforward. You can define a shortcut for the common case of adding a dated report by using the current username. You don't have to bother to set the `date_fixed` field because the `who_fixed` member is set to zero, and this will mean "still pending, not fixed yet":

```
void add_report(string who, int level, Date date,
                string descript)
{
  Report report;
  // fill in the report structure
  report.id = next_id();
  report.level = level;
  report.date_reported = date;
  set_description(report,descript);
  report.who_reported = Users::id(who);
  report.who_fixed = 0;  // meaning, not fixed yet!

  ofstream out(BUGFILE,ios::binary | ios::app);
  out.write((char *)&report,sizeof(Report));
}

void add_report_today(int level, string descript)
{
  add_report(Users::myself(),level,Dates::today(),descript);
}
```

The following are the routines for reading through the bug report file and printing reports out if they match the criteria:

EXAMPLE

```
void dump_report(ostream& out, const Report& report)
{
    out << report.id << ' ' << report.level
        << ' ' << Users::name(report.who_reported)
        << ' ' << Dates::as_str(report.date_reported)
        << ' ' << get_description(report) << endl;
}

void list_reports(ostream& out, int min_level, Date d1, Date d2)
{
  Report report;
  ifstream in(BUGFILE, ios::binary | ios::in);
  while (in.read((char *)&report,sizeof(Report)))
    if (report.level > min_level &&
            Dates::within_range(report.date_reported,d1,d2))
      dump_report(out,report);
}

bool read_report(int id, Report& rpt)
{
  ifstream in(BUGFILE, ios::binary | ios::in);
  in.seekg((id-1)*sizeof(Report));
  if (!in.eof()) {
       in.read((char *)&rpt, sizeof(Report));
       return true;
  } else return false;
}
```

In this example, you separate out the code that does the dumping to simplify `list_reports()`. As you can see, it is considerably simpler than the equivalent function using text files; the whole structure is read by using the `read()` method, which returns `false` when the end of the bugs file is reached. The function read_report() returns a report from the file, given its ID. Assuming that the IDs are consecutive integers, it is straightforward to go to exactly the position where you can read the requested report.

The code for write_report() is very similar to the code for read_report(), except that you use seekp() rather than seekg() and write() rather than read(). (See chap5\bugs.cpp on the CD-ROM that accompanies this book.) Being able to modify a bug report is a powerful feature; with the old text file representation, you would have had to rewrite the file. The question of deleting bug reports doesn't occur with this version of the bug tracker, because even a false report must be dealt with and written off as "not-a-bug" and kept for future reference. But if you did want to delete bug reports, it would be easier to mark them as being deleted than to rewrite them each time. This technique is often used in implementing databases; at some future point, you compact the database, which actually removes all the collected garbage by leaving out records marked as deleted.

The Utility Program Interface

The prototypes of the functions in bugs.cpp are collected in bugs.h, and you can now pull this together as a program:

EXAMPLE

```
// addbg.cpp
#include <iostream>
#include <cstdlib>
using namespace std;

#include "bugs.h"

int main(int argc, char **argv)
{
// remember, argc counts argv[0] as well!
  if (argc != 3) {
      cerr << "usage: <level> <description>\n"
           << "Adds a bug report to the database\n";
      return -1;        // this program failed!
  }
  int level = atoi(argv[1]);
  if (level == 0) {
      cerr << "Bug level must be > 0\n";
      return -1;
  }
  Bugs::add_report_today(level, argv[2]);
  return 0;  // success
}
```

It is considered bad manners for programs to crash because of bad input, so you should try to give users sensible error messages.

To build this program, you have to compile all the source files and link them together. Initially, the following DOS command will compile all the files, and subsequently you have to recompile only the files that change. Refer to Appendix D, "Compiling C++ Programs and DLLs with GCC and BCC32," for the options, including setting up a project in an integrated development environment such as Quincy 2000.

```
C:\bolsh\ucw\chap5> c++ addb.cpp bugs.cpp users.cpp dates.cpp - o addbg.exe
```

Extensions

Let's now work on the case where the description is more than `MAX_DESCRIPTION` characters. Fortunately, there are two extra fields available; these fields cannot save more than eight characters, but one of them can give an offset into another file. That is, if there are too many characters, you can append them to the end of a file of strings and record the position in the `extra1` member of the struct. There is some redundancy because you save the first `MAX_DESCRIPTION` characters twice—in the bug report file and also in the file of strings—but that would not be difficult to improve. The following code shows new definitions for `set_description()` and `get_description()`.

Writing out the description to the strings file is straightforward because you aren't forcing the string into a fixed-length block of characters. The string's raw character data can be written directly to the file as so many bytes. You can 'tell' what the offset at the end of the file is by using `tellg()`. Before you write, you save this offset.

Reading the string back involves moving to the stored offset with `seekg()` and reading `descript_size` bytes into a temporary buffer, which is then returned as a `string`. `get_description()` declares this buffer as `static` because sometimes too much local storage can cause a stack overflow.

EXAMPLE

```
void set_description(Report& report, string descript)
{
 int sz = descript.size();
 report.descript_size = sz;
 strncpy(report.description, descript.c_str(), MAX_DESCRIPTION);
 // extra long description?
 if (sz > MAX_DESCRIPTION) {
   ofstream out(STRSFILE,ios::binary | ios::app);
   report.extra1 = out.tellp();
   out.write(descript.c_str(),sz);
 }
}

string get_description(const Report& report)
{
 if (report.descript_size < MAX_DESCRIPTION)
      return report.description;
 else { // read the description from the strings file
   static char buffer[TRULY_MAX_DESCRIPT];
   ifstream in(STRSFILE,ios::binary);
   in.seekg(report.extra1);
   in.read(buffer,report.descript_size);
   return buffer;
 }
}
```

Note that you could make these modifications to the program without having to change the rest of the program because the description reading and writing logic has been separated out. That is, you are just replacing `set_description()` and `get_description()`.

Later in this book, we'll explore this issue in greater detail, but it's useful to note that object-oriented programming is more of a mental habit than using object-oriented language features like classes. With object-oriented programming, you need to use *encapsulation* (that is, wrap up tricky logic and assumptions into functions, which are given exclusive right to manage some data). It is possible to write object-oriented programs in any language that has structures and functions. And conversely, you could write incoherent, bad code and dress it up in classes. I will introduce the object-oriented features of C++ like classes in Chapter 7, "Classes."

What's Next

This chapter introduced the important ideas of user-defined structures and pointer types. Together with dynamic allocation, they allow you to generate flexible data structures of any desired complexity, like linked lists.

The next chapter goes into more detail about C++ functions. A number of functions may share the same name but have different arguments (called *function overloading*). Since C++ operators are essentially functions, they may also be overloaded for specific functions. And sometimes it is very useful for functions to be able to call themselves. I will do two graphics examples that show this in action.

6

Overloading Functions and Operators

C++ has many features that make life easier for programmers. For example, thinking up unique and meaningful names for functions can be irksome. You saw in Chapter 4, "Programs and Libraries," how namespaces can keep families of functions together, but even with namespaces, you still need to come up with names such as `print_date()` and `print_person()`. With the C++ feature *function overloading,* you can let one name stand for a number of functions, depending on the arguments. In C++ you can also redefine operators, so you can define a simple notation for common operations on structures. Functions can call themselves, under special conditions, and they can be passed as parameters to other functions.

In this chapter you will learn about

- Overloading functions and using default parameters
- Redefining C++ operators, especially for output and input
- Recursive algorithms
- Using Turtle Graphics

Default Values for Parameters

To make a function as general as possible, you need to specify all its parameters. It is often a bad idea to make a function's result depend on some global parameter, and it is frustrating to use a function that depends on some mysterious constants.

Specifying too many parameters as arguments is clumsy and makes the function's use hard to remember. In Chapter 5, "Structures and Pointers," you saw one solution—to wrap up the parameters as members of a structure, which could then be efficiently passed by reference to a function. This solution is the equivalent of the named parameters that some languages support. Here is an example of what this kind of parameter passing looks like. `do_operation()` is a function with a large number of parameters, but we want to change only the font used for text output. Instead of passing a dozen or more parameters, only the font properties are changed. Again, this technique can be clumsy because you usually need to initialize the other members of the structure with some default values.

```
;> OperationArgs args;
;> args.flags = SET_FONT;
;> args.fontsize = 12;
;> args.fontname = "arial";
;> do_operation(args);
```

You can use default arguments to handle this problem. For instance, here is a function that prints a little table of sine and cosine values to some output stream. Notice that the last argument `dx` of `dump_trig()` is declared like an initialized variable (`double dx = 0.1`). Any arguments that specify initial values in this way are called *default arguments*. Such arguments may be left out of function calls.

EXAMPLE

```
;> void dump_trig(ostream& out,
;>      double x1, double x2, double dx = 0.1)
;> {  for(double x = x1; x < x2; x+=dx)
;2}     out << sin(x) << ' ' << cos(x) << endl;
;1} }
;> dump_trig(cout,0.0,0.5);  // only out, x1, and x2 specified!
0.000000 1.000000
0.099833 0.995004
0.198669 0.980067
0.295520 0.955336
0.389418 0.921061
```

In this example, the function `dump_trig()` defines a sensible default value for the gap between `x` values `dx`. If you choose not to use the last argument, a value of 0.1 will be assumed, or you can specify all four arguments.

In the previous chapter, you defined the `make_point()` function to construct `Point` objects. Here it is redefined to use default arguments:

```
Point make_point(int x=0, int y=0) {
   Point p;
   p.x = x;  p.y = y;
   return p;
 }
;> Point p1 = make_point();
;> p1.x;  p1.y;
(int) 0
(int) 0
```

In this example you can specify two, one, or no arguments for `make_point()`. It is important to have sensible defaults that would be obvious (for example, setting the point coordinates to `(10,10)` would not be obvious). Zero is a good obvious value for many types.

NOTE

It is an error to redefine default arguments in the function definition if the function definition has already been declared with default values. So, if your function is defined in a header, keep the default values there.

Overloading Functions

A powerful feature of C++ is the ability to use the same name for several functions. This sounds like a confusing thing to do, but when people **draw** water and **draw** a card the meaning of 'draw' is usually obvious from the context. In the same way, C++ uses the argument types to distinguish between the various *overloaded functions*.

sqr()

The `sqr(x)` function squares its argument. This operation makes sense for any argument that you can multiply by itself (for instance, both floating-point and integer numbers). Bear in mind that floating-point and integer arithmetic are very different on a machine level, and using a `double sqr(double)` function to do integer squaring can be very inefficient. So you need to define two functions, `sqr()` and `sqr_int()`, but the marvelous thing about C++ functions is that you can use the same name for both functions. Here's how it works:

```
;> double sqr(double x) { return x*x; }
;> int sqr(int I) { return I*I; }
;> sqr(2.0);
(double) 4.
```

```
;> sqr(2);
(int) 4
```

`double sqr(double)` and `int sqr(int)` are two very distinct functions; their implementation may look similar, but floating-point multiplication is different from integer multiplication, even though you use `*` for both types. The operator `*` is in fact itself overloaded. The compiler can distinguish the correct function to use because the `double` argument is distinct from the `int` argument. We say that the *signature*—that is, the set of arguments plus the return type—of the functions is different. The correct function is resolved from the overloaded set of functions with the same name. And you can continue to add functions called `sqr()` to the *overloaded set* providing their arguments are sufficiently different. For example, some like to define *squaring of a `vector`* to mean the `vector` of squared values:

```
;> typedef vector<double> Dvect;
;> Dvect sqr(const Dvect& v) {
;1} Dvect tmp = v;
;1} for(int k=0,n=v.size(); k < n; k++)
;2}   tmp[k] = sqr(tmp[k]);
;1} return tmp;
;1} }
```

It appears as if this version of `sqr()` is defined in terms of itself, but this isn't so: `sqr(tmp[k])` has an argument of type `double`, which matches `sqr(double);` and that is a different function. This appears marvelous, but there is a downside. First, if there are separate `sqr()` functions for `int` and `double`, you cannot depend on the usual `int`-to-`double` conversion: The system picks the best fit and doesn't try to force the `int` argument into a `double` function. You can always force the function you want by explicitly writing floating-point numbers (for example, `2.` or `2.0`) or by using a typecast (for example, `sqr((double)k)`). Also, the signatures need to be sufficiently different from one another. If the compiler finds two or more functions that match equally well, an error results. Say that you defined the floating-point `sqr()` by using `float`, not `double`:

```
;> float sqr(float x) { return x*x; }
;> sqr(1.2);
CON 4:Ambiguous match for void sqr(const double)
    int sqr(int)
    float sqr(float)
```

To begin with, a constant like `1.2` has type `double`. People tend to instinctively feel that `float` is somehow closer to `double` than it is to `int`, but this is not necessarily so. It is true that a `float` will be promoted to a `double`, and the system will prefer this to doing a standard conversion from `int` to `double`. But both `double`-to-`int` and `double`-to-`float` are narrowing operations in which information could be lost (often called *demotion*) because 8 bytes of precision is being forced into 4 bytes. So `double`-to-`int` and `double`-to-`float` are both considered standard conversions and are equally favored. Both versions of `sqr()` will then match a `double` argument, and this is called an *ambiguous match*. You should prefer `double` arguments to `float` arguments for this reason.

An ambiguous match error is not actually a bad thing; it is worse when the system silently picks the wrong function without any warning. (Error messages are very useful when it comes to getting programs right, and **everyone** gets compile errors when writing serious programs.) These potential problems may make you feel nervous about using overloaded functions. However, you don't need to know all the rules to use overloaded functions effectively. Keep the argument types as distinct as possible, and for the most part, things should behave themselves.

Functions can be distinguished by the numbers as well as by the types of their arguments. Distinguishing them by number is the best way of keeping functions distinct and unambiguous. For example, these are clearly different signatures because they have a different number of arguments:

```
;> int minv(char arr[], int sz);
;> int minv(char ptr[]);
```

Different Parameters for the Same Operation

The basic question to ask when naming a function is this: Does this name make the function's purpose clear to some future user? If it is difficult to name a function, then maybe the function's purpose is confused anyway; perhaps the function needs to be broken up into two or more distinct operations. This is particularly true when functions are overloaded. You reuse names when a name describes a distinct operation that is applicable to different arguments. You could probably write a whole program in which all the functions (except for `main()`, of course) were called `function()`, and this would be a bad thing not only because the names are not descriptive but because there is no common operation. Generally, it's a good idea to keep names distinct.

An Alternative to Default Values

Sometimes it is unnecessary to overload functions because default arguments will do the job better than overloaded functions. A good example is the `make_point()` function discussed earlier in this chapter. You could achieve the same effect as that `make_point()` function by using two functions, declared like this:

```
;> Point make_point(int x, int y);
;> Point make_point();
```

If you supply definitions for these functions, you will see that the two implementations are practically the same—they aren't really different functions at all. You can use both default arguments and overloaded functions, but be aware that they sometimes collide with each other. If you supplied default arguments for the first function in the preceding example, the overload resolution would become ambiguous because there are two identical ways of matching the same signature:

```
;> Point make_point(int x=0, int y=0);
;> Point make_point();
;>  make_point();
CON 16:Ambiguous match for void make_point()
    Point make_point(int,int)
    Point make_point()
```

Of course, you can change your mind in the future about whether to use default arguments or function overloading because the effect is the same. Generally, you should use default arguments if the implementation of the other case would be almost identical, like with the preceding two versions of `make_point()`. But if the implementation is different, then overloading would be better. You should use overloading only if the functions have very different ways of doing the same thing; for example, `print(Point)` and `print(Date)` would be good candidates.

Overloading Operators

Operators such as `+` and `*` in C++ can be considered to be functions taking two arguments (these are often called *binary operators*). Because they are functions, they can be overloaded. This section shows why and how you can overload operators.

Adding Two Points Together

Points in the two-dimensional space of a graph can be added together by adding their coordinates. They can also be divided by a *scalar* (that is, a single number). For example, you calculate the average of two points by adding them together and dividing by 2. The result is a point that is midway between them in space. This shows how the average of two `Point` structures can be calculated, by defining `add()` and `div()` functions:

```
Point add(Point p1, Point p2) {
   Point p;
   p.x = p1.x + p2.x;
   p.y = p1.y + p2.y;
   return p;
 }
Point div(Point p, int val) {
   p.x /= val;
   p.y /= val;
   return p;
 }
;> Point p1 = make_point(1,4), p2 = make_point(5,2);
;> Point p = div(add(p1,p2),2);
;> show(cout,p);  // print out the average of (1,4) and (5,2)
 (3,3)
```

With function overloading, you don't have to worry about naming these functions `add_point()`, `show_point()`, and so on. But it would be nice to use the usual arithmetic notation, not the function notation. That is, people are more used to writing `(p1+p2)/2` than writing `div(add(p1,p2),2)`. Functional expressions like this are often more difficult to read, and bracket-counting is necessary. The next section shows how to use mathematic notation in this case.

Operators as Functions

C++ operators are functions that can be redefined. Here are overloaded versions of the addition and division operators that work with `Point` types:

EXAMPLE

```
Point operator+(Point p1, Point p2) {
   Point p;
   p.x = p1.x + p2.x;
   p.y = p1.y + p2.y;
   return p;
 }
Point operator/(Point p, int val) {
  return div(p,val);
}
;> Point p1 = make_point(1,2), p2 = make_point(5,2);
;> Point p = (p1 + p2)/2;
;> show(cout,p);
 (3,3);>
```

The definition of `operator+()` is precisely the same as a normal function definition, except that you use the keyword `operator` followed by the symbol (for example, `operator/`). It is now possible to write point arithmetic precisely like normal arithmetic! Java programmers call this "semantic sugar," but people are very used to normal arithmetical notation, and it's easier to get right and (just as important) easier to read than functional notation. Just because sugar makes some programs (and people) fat doesn't mean we have to abandon sugar altogether.

Careful: The fact that you have redefined + and / for points doesn't mean that += and /= are automatically redefined. C++ does not check that your definitions for those operators are consistent: It's up to you to ensure that p=p+q is the same as p+=q. operator+= is a good idea for big structures because it can be defined entirely with references. This operator is passed two reference arguments, the first of which is not const because it will be modified by the operation. The following code defines an operator+= that returns a reference to the modified point, and no copying is involved. Bear this in mind if fast and compact code is essential (that is, you have to watch your sugar).

```
Point& operator+= (Point& p1, const Point& p2) {
  p1.x += p2.x;
  p1.y += p2.y;
  return p1;
}
```

There are some things you can't do with operator overloading, and there are some things you shouldn't do. You cannot redefine any C++ operator for the usual simple types, such as double and int. It would be very dangerous for people to be able to modify basic arithmetic. (Sometimes you need unlimited-precision arithmetic, but in those cases, you can define special structures and overload the arithmetic operators appropriately.) Some operators such as the member selection, or dot (.), operator can't be overloaded at all. You can't define your own operators, such as $, and you can't change the precedence of the operators (for example, operator* will always have a higher precedence than operator+). These rules prevent people from completely redefining the language to suit their own (possibly weird) tastes.

Obviously, you would not define - to mean addition, but the overloaded operator must still be meaningful. Say you had a Club struct and defined += so that it took a Person operand and had the effect of adding Person to Club. This would be silly because this addition here has nothing to do with mathematical addition. (+= does also mean string concatenation, but that is a well-established notation.) Naming functions requires thought, and deciding which operator to use requires particular consideration. Operator overloading is very powerful, but it can be abused.

Overloading << and >>

The insertion operator (<<) can be redefined similarly to other operators. In fact, it is more commonly used in its overloaded form than in its basic form. The original meaning of operator<< is to do *bit shifting*. Given the binary representation of an integer, << shifts the bits to the left by the specified number of places. It is equivalent to multiplication by powers of 2. So, for example, 10 << 1 would be 20.

Consider the business of building up a list `ls` of integers. Typing `ls.push_back(i)` can be tedious. It is convenient to redefine `operator<<` to mean "*add to list*"; I've simplified the definition of this operator by using the `typedef` symbol `LI` to mean `list<int>`:

EXAMPLE

```
;> typedef list<int> LI;
;> LI ls;
;> LI& operator<< (LI& ls, int val) {
;1}  ls.push_back(val);
;1}  return ls;
;1}  }
;> ls << 1 << 2;
(list<int>&) list<int> {}
;> ls.front(); ls.back();
(int&) 1
(int&) 2
```

`operator<<` is usually defined to return a reference to the first argument. This makes the preceding trick possible. `ls << 1 << 2` can be written as `(ls << 1) << 2`. `ls<<1` is evaluated first, and the result is `ls`, and so finally `ls<<2` is evaluated. It is exactly the same as typing `ls << 1; ls << 2;`.

By far the most common use of overloading << is for output. This way of chaining << and values is how C++ stream output is done. You can overload `operator<<` for `Point` arguments like this:

```
ostream& operator<<(ostream& os, Point p) {
    os << '(' << p.x << ',' << p.y << ')';
    return os;
 }
;> cout << "the point is " << p1 << endl;
the point is (1,4)
```

The resulting operator can be used in any output expression. It operates on a general `ostream`, so you can use it for file output or writing to strings. This is exactly how the `<iostream>` I/O stream libraries are built up: by overloading `operator<<` for the basic types. (Look at the end of the `<iostream>` header in your UnderC `include` directory to see this in action.) A powerful feature of this library is how easy it is to extend it for user-defined types.

Similarly to overloading the << operator, you can overload the extraction (that is "get from") operator for `Point` arguments:

```
istream& operator>>(istream& is, Point& p) {
    is >> p.x >> p.y;
    return is;
 }
;> Point p;
;> cin >> p;
100 200
;> cout << p << endl;
(100,200)
```

Note that you must pass a reference to `Point` so that it will be modified by the input operation. Be careful about this, because forgetting the `&` leaves you with an operator that changes a local variable and nothing else. Again, operator>> operates on a general input stream.

Recursion

Recursion is when a function calls itself. This seems like a recipe for disaster, but recursion is a powerful way to express some operations that are naturally defined in terms of themselves.

Another Binary Search

Consider the binary search introduced in Chapter 3, “Arrays and Algorithms”: You look at the midpoint of the range, and if that value equals the key, then you have succeeded. If the key is less than the middle value, you search the first half, and if the key is greater than the value, you search the second half. Finally, if the range is empty, you have failed. Note that this description of a search is defined in terms of searching! Here is a version of `bsearch()` that has no explicit loop. Instead, it calls itself to search either the first or the second half:

EXAMPLE

```
int bsearch(int arr[], int low, int high, int val)
{
    int mid = (low+high)/2;
    if (val == arr[mid]) return mid;  // success
    if (low == high)     return -1;   // failure
    if (val < arr[mid])               // keep searching…
       return bsearch(arr, low,  mid-1,val); // first half
    else
       return bsearch(arr, mid+1,high, val); // second half
}
```

Recursion must terminate at some point. Eventually there must be some simple and unambiguous case that can be evaluated immediately. In this case, either you have found the key, or the interval (`low`, `high`) is empty.

How does this magic occur? Consider the list `{4,6,10,15,20,30}` and the key `6`. You can start with `low=0` and `high=5`, and the midpoint index is `3`; the value there (`15`) is greater than 6, so you call `bsearch()` with `low=0` and `high=3`. The midpoint is at `1`, where the value is exactly right; a hit!

Why Factorial Isn't So Cool

A classic example of a recursive function is the factorial function. Factorials are very common in statistics and are related to the number of ways you can arrange a number of objects. The factorial of 4, which is written 4!, is defined as $4\times3\times2\times1 = 24$. 0! is defined to be 1; otherwise, $n! = n\times(n-1)!$; the factorial function can be defined in terms of itself. So you can write a recursive function to work out the factorial as follows (note that it continues until the stopping condition, which is the definition of 0! as 1):

```
long fact(int i) {
  if (i == 0) return 1;
  else return i*fact(i-1);
 }
;> fact(4);
(long int) 24
;> fact(6);
(long int) 720
```

This function will go haywire with negative values for the argument, because `i` will continue to be decremented and will never be equal to zero. It will continue to call itself, but until when? Not forever; each call to `fact()` uses up a little more of the *runtime stack*. In most implementations of C++, the runtime stack is used for three purposes: saving the return address, passing parameters, and allocating temporary space for local variables. When a function is called, the arguments are pushed onto the stack. Then the current address in the program (often called the *instruction pointer*) is pushed. When the function is entered, the stack is further pushed by the number of words needed to store local variables. When the function is left, these temporary words are popped, the return address is popped, and finally the arguments are popped. This is why local variables cease to exist after their scope is closed. (This is also why it is important not to overwrite local arrays in functions; it usually corrupts the stack by destroying the return address, and the program crashes in confusion.)

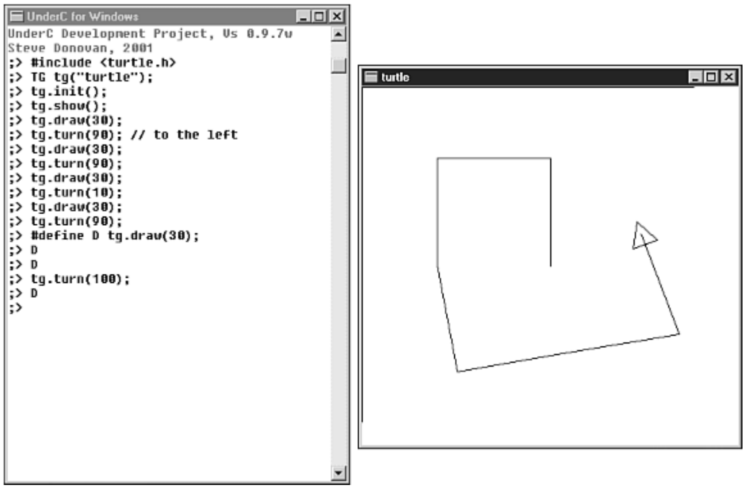
Recursion can put heavy stress on the runtime stack, particularly if you have declared a lot of local storage. Because calling a function is more expensive than just looping, it is a good idea to use recursion only when it is genuinely necessary. Factorials can be done with straightforward loops, and in fact people don't usually evaluate them directly because they overflow so quickly (that is, they rapidly become larger than the largest possible 32-bit integer.)

Drawing Trees with Turtle Graphics

In the late 1960s at MIT, Seymour Papert and others conducted an interesting experiment. They exposed kids to serious computers for the first time, even inventing a programming language (LOGO) for children in the process. Papert's intention was to create a Mathworld in which learning mathematics was easier, in just the same way that it is easier to learn French if living in France. LOGO was an intensely graphical environment and introduced the idea of the turtle, either on the screen or actually on the floor. The *floor turtle* was a little robot which could be made to turn at any angle and move forward and backward. If a pen was attached to it, it would draw pictures on the floor. A *computer turtle* is a little triangle in a graphics window that behaves like a floor turtle. An educational benefit of drawing with turtles is that children can identify with them; with LOGO, if you wanted to know how to draw a square, you would pace it out on the floor yourself and then type in the instructions. You would then define a new word (what we would call a function) and use it as a building block. Turtle graphics are very powerful for drawing shapes compared to using Cartesian coordinates (that is, using x and y coordinates). Some patterns, as you will see, are much easier to draw by using this method. This implementation of turtle graphics is specific to UnderC, although I have included the necessary files to build such applications using GCC and BCC32 (please see Appendix D for more information.)

In the following example, you first have to `#include` the system file `<turtle.h>`. Then you can construct a turtle graphics (TG) window with the declaration `TG tg("turtle")`, where the string constant will be the window caption. The `init()` method is needed to initialize this window, and the `show()` method to show the turtle. At this point you should see the turtle pointing up in the middle of the new window. The `draw()` method causes the turtle to advance. After you type `tg.turn(90)`, the turtle points to the left. Any further calls `draw ()` to continue the line in the direction that the turtle is pointing. You can switch off the turtle by using `tg.show(false)`, but it is useful to see the turtle at first because it reminds you that a turtle has both position and direction. Figure 6.1 shows the turtle in action.

```
;> #include <turtle.h>
;> TG tg("turtle");
;> tg.init();
;> tg.show();
;> tg.draw(30);
;> tg.turn(90);
;> tg.draw(20);
```



```
UnderC Development Project, Vs 0.9.7v
Steve Donovan, 2001
;> #include <turtle.h>
;> TG tg("turtle");
;> tg.init();
;> tg.show();
;> tg.draw(30);
;> tg.turn(90); // to the left
;> tg.draw(30);
;> tg.turn(90);
;> tg.draw(30);
;> tg.turn(10);
;> tg.draw(30);
;> tg.turn(90);
;> #define D tg.draw(30);
;> D
;> D
;> tg.turn(100);
;> D
;>
```

Figure 6.1: *The turtle in action.*

In an interactive session, typing these commands can become tedious. The C++ preprocessor can help you define suitable macros to avoid some of the typing. Here is how the `#define` directive can save typing:

```
;> #define L tg.turn(+90);
;> #define R tg.turn(-90);
;> #define F tg.draw(10);
;> F L F R
```

A symbol that has been defined (using `#define`) is replaced by its definition, including the final semicolon. So, for example, `L` expands to `tg.turn(+90);`.

TIP

The preprocessor got a bad reputation thanks to the abuse by C cowboys, so many people dislike all macros. But in this section we're talking about a perfect use for them; in an iterative session, the ability to create abbreviations is very useful. However, they are usually not appropriate in a proper program, especially one that has to be publicly accessible. Expressions such as *ain't* are acceptable in colloquial speech but not in business letters. See Appendix C, "The C++ Preprocessor," for a discussion of the preprocessor and its enemies.

The state of the turtle at any point is its orientation and position. The type `TG_State` can be used to store this state for later. In the following example, you reset the turtle with `setup()` and create a variable of type `TG_State`:

```
;> tg.setup();
;> TG_State state(tg);
;> tg.draw(2);
;> state.save();
;> tg.turn(45);
;> tg.draw(2);
;> state.restore();
;> tg.turn(-45);
;> tg.draw(2);
```

In this example, you draw a line, save the state, turn right and draw, restore the state, and turn left and draw. (Type these commands to see their effects for yourself!) The result is a Y pattern. Now consider how to draw this Y using Cartesian coordinates; it's not too difficult. But if you had turned the turtle by an angle before the last set of commands, then the Y would have been at an angle. That is, any shape drawn with turtle graphics can be easily rotated by setting the initial orientation. And this is genuinely tricky to do with conventional plotting calls.

You can generate interesting patterns by using recursion. The following recursive function draws a tree:

EXAMPLE

```
void draw_tree(TG& tg, double l, double fact,
            double ang=45.0, double eps=0.1)
{
 TG_State state(tg);
 if (l < eps) return;
 //---- Draw the line
 tg.draw(l);
 //---- Save state, turn right, and draw the right branches
 state.save();
 tg.turn(ang);
 draw_tree(tg,fact*l,fact,ang,eps);
 //---- Restore state, turn left, and draw the left branches
 state.restore();
 tg.turn(-ang);
 draw_tree(tg,fact*l,fact,ang,eps);
}
```

Again, the stopping condition is the first thing you check, and this is when the length (`l`) passed to the function is less than some small number (`eps`). Note that the left and the right sides of the tree are drawn by calls to `draw_tree()`, but using a length of `l*fact`, where `fact` must be less than `1.0`. So eventually, `l` will be small enough, and the recursion will end. In practice, this function can go wild if you aren't careful. If you launch this function from `main()` using UnderC and use #r to run the program, then #s can be used to stop the program at any point.

Note that the function saves the turtle graphics orientation and position in `state`, which remembers where you started. The call to `draw_tree()` also defines a variable, `state`, but (and this is the crucial bit) this is distinct from the last values. Useful recursion is possible only because automatic variables are allocated on the stack, which is why languages such as FORTRAN and BASIC traditionally couldn't handle recursion.

Figures 6.2 and 6.3 show the result of calling `draw_tree()` with two different angles of branching.

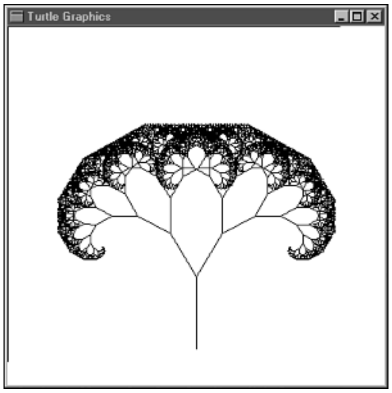


Figure 6.2: *The result of calling* `draw_tree(20,0.7,30,0.1)`.

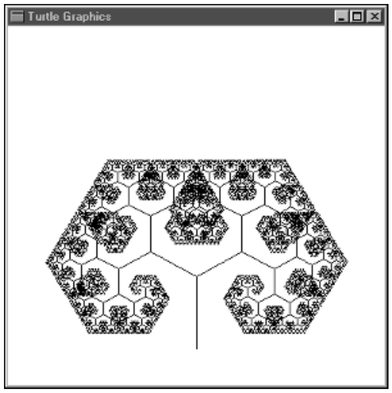


Figure 6.3: *The result of calling* `draw_tree(20,0.7,60,0.1)`.

The power of using recursion to model biological trees comes from the nature of growth. The genes of the tree don't determine each twig; they determine the twig-sprouting recipe, and they supply local rules that generate the pattern.

Function Pointers

All C++ code is translated into some form of machine code, which resides in memory somewhere when a program is running (in the case of UnderC and Java, the machine code is for a virtual machine that runs on top of a real machine). So code is a form of data, and thus it has an address. Usually the operating system organizes it so that code occupies protected memory that can't be written to, but it can be read and executed. Applying the address-of operator (&) to a function reveals this address:

```
;> &sin;  &cos;
(double operator(*)(double)) 72B364
(double operator(*)(double)) 72B390
```

This function address can be kept in a special kind of pointer, which can be used as a function. The dereference operator (`*`) is used to extract the value of the pointer. The function pointer declaration in the following code may seem strange at first. It is similar to a function prototype declaration, except for the `(*)`. The parentheses are necessary because otherwise it would simply be a declaration, `double *pfn(double)`, which is just a function taking a `double` argument and returning a pointer to a `double`. Once `pfn` is declared, it can be assigned the address of any compatible function. `pfn` can then be used to call the function using a `double` argument:

```
;> double (*pfn)(double);
;> pfn = &sin;
;> (*pfn)(1.0);
(double) 0.841471
```

Function pointer declarations are awkward, so usual practice is to define a `typedef`. As usual, the `typedef` looks exactly like a variable declaration, except the variable becomes a type alias:

```
;> typedef double (*DFN)(double);
;> DFN p1 = &sin, p2 = &cos;
;> p2(1.0);
(double) 0.540302
```

Note that you don't have to use the dereference operator here because it's clear from the context that this is a function call. The address-of operator is also implied when a function appears without being called. This is different from the usual pointer behavior; you also cannot freely convert function pointers to `void` pointers, because you usually cannot treat function pointers as data.

You can use function pointers to pass functions to other functions. A classic example of this would be code that plots a given function, such as the following:

EXAMPLE

```
void vplot(TG& g, DFN pfn,
   double x1, double x2, double dx=0.1)
{
// construct a vector containing the function values
  int n = (x2 - x1)/dx;
  vector<double> v(n);
  int i = 0;
  for(double x = x1; x < x2; x += dx)
     v[i++] = pfn(x);

// find our min/max data values
  vector<double>::iterator v1 = v.begin(), v2 = v.end();
  double vmin = *min_element(v1,v2);
  double vmax = *max_element(v1,v2);

// scale our graphics and plot out the points….
  g.scale(x1,x2,1.5*vmin,1.5*vmax);
  g.init();
  g.penup();
  for(double x = x1; x < x2; x += dx)
     g.plot(x,*v1++);
}
```

An example of using this would be `vplot(tg,0,10,&sin)`. (Note the default argument.) This example uses the standard algorithms `min_element()` and `max_element()` (you must include `<algorithm>` for these), which return iterators, not values (hence the dereference, `(*)`). The remainder of the example shows how you can do plain Cartesian graphics with a turtle graphics window. `TG::plot()`draws a line to the given point, unless the pen is up. After a call to `plot()`, the pen is always down. (This style is less hassle than needing different calls, `move_to()` and `draw_to()`, to draw lines. It comes from the days when you really could see the pen moving about on the plotter.) You can of course pass any function, not just standard library functions, provided that the function has the correct signature, `double fn(double)`.

Several standard library algorithms rely on function pointers. It is important to avoid writing out loops as much as possible. For example, say you want to print out a list of integers. Previously you would have declared a suitable iterator and incremented that iterator from `ls.begin()` to `ls.end()`. The standard algorithm `for_each()` does the job without needing the clumsy iterator declaration. For each element in the sequence, `for_each()` calls the specified function. The algorithm `transform()` is similar, except that it modifies the sequence by saving the result of applying the function to each element. Keep in mind that `transform()` doesn't necessarily write the result back to the sequence. As with `copy()`, you can use `transform()` to modify another, different, output; but often the output is the same as the input. In the following example, you set any negative numbers in the sequence to zero and use the library function `toupper()` to change each character in a string to uppercase:

```
;> list<int> ls;
;> ls << 10 << -20 << 30;   // using operator<< defined previously!
;> void print_int(int i) { cout << i << endl; }
;> for_each(ls.begin(), ls.end(), print_int);
10
-20
30
;> int positive(int i) { if (i < 0) return 0; }
;> transform(ls.begin(),ls.end(),ls.begin(),positive);  // ls is now 10 0 30
;> string s = "hello";
;> #include <cctype>  // for toupper()
;> transform(s.begin(),s.end(),s.begin(),toupper);
;> s;
;> (string) s = "HELLO"
;> #define ALL(s)  s.begin(),s.end()
;> transform(ALL(s),s.begin(),tolower);
;> for_each(ALL(ls),print_int);
10
0
30
```

This example uses the `#define` macro `ALL`, which takes one parameter that will be substituted when the macro is replaced (or expanded). This makes experimenting with these algorithms more fun because it saves having to type (and possibly get wrong) the begin/end sequence. (You should not use such shortcuts in official programs.)

Function pointers behave like ordinary pointers in many ways; you can construct arrays, lists, vectors, or maps, using them. Here is an array of pointers to trigonometric functions (note again that the dereference isn't necessary when calling these functions):

```
;> double my_fun(double x) { return sin(x*x); }
;> DFN funs[] = { &sin, &cos, &tan, &my_fun };   // an array of function pointers
;> (*funs[0])(1.0);  funs[0](1.0);
(double) 0.841471
(double) 0.841471
```

Arrays of function pointers make general menu code easy and flexible. After all, without such things, you are forced to write an explicit `switch` statement, which is impossible to extend at runtime. That is not an option for the function plotter because it must plot any function. Maps of function pointers can be particularly powerful. For instance, in the reverse-Polish calculator case study in Chapter 4, “Programs and Libraries,” you could have kept a map of strings to function pointers.

Case Study: Drawing Shapes with Turtle Graphics

Graphics programming is a very satisfying activity, because you get immediate visual feedback. Most introductory C++ books don’t do graphics for the simple reason that there are no standard graphics calls, unlike with Java. However, the principles are usually the same. Doing some graphics is (a) enjoyable and (b) better than doing no graphics at all.

This case study shows how structures can be used to store information about shapes, and how then these shape objects can be drawn using turtle graphics in more detail. At the end, there is some more advanced but optional material.

The Specification

You need to write a program that takes a file describing shapes and draws the shapes on a window. Initially, these shapes are squares, rectangles, circles, ellipses, and lines; color can be specified for both lines (foreground) and shapes (background). Here is an example of the proposed format:

```
RECT    10 10 30 40
SQUARE  40 20 10
CIRCLE  80 20 10
FCOLOR  255 0 0
BCOLOR  0 0 255
ELLIPSE 80 40 120 60
INC STARS 200 200
```

General shapes such as rectangles and ellipses are specified by their bounding rectangle, whereas squares and circles are specified by an origin followed by a size. Colors are specified by three numbers—red, blue, and green—each of which varies from 0 to 255. This gives you 3 bytes, or 24 bits, per color. There is also an `include` command that brings in other plot files; this can be (optionally) drawn in a specified location.

The Representation

Your first design decision is to read the file into a data structure, which is then used for doing the actual plotting. There are a number of good reasons for this. First, it is efficient because the file will be read only once, and the shapes will have to be drawn whenever the window repaints itself. Second, you need to find the minimum and maximum bounds of the shapes, which would not only require another reading of the file but would be awkward.

Third, it allows you to split the problem neatly into two parts—reading in and plotting out—and this will simplify development.

A `struct` would be the best way to represent each shape. You could use unstructured arrays of numbers, but as many a FORTRAN programmer has discovered, this ends up confusing both the guilty and the innocent. Shapes naturally can be specified by their bounding rectangles (that is, the smallest rectangle than encloses the shape); as far as the representation goes, a circle is merely an ellipse with a square bounding rectangle. If it is later important to know the difference between circles and ellipses, it is easy to compare the dimensions of the rectangle.

The first struct you need to create is `Rect`, which consists of `left`, `bottom`, `top`, and `right` (with the `typedef real` defined as the chosen floating-point number):

```
typedef unsigned char byte;
typedef double real;
typedef unsigned long Color;

struct Rect {
  real top,left;
  real bottom,right;
};

enum Type {ELLIPSE,RECTANGLE,LINE};

struct Shape {
   Type type;
   Rect rect;
   Color fcolor,bcolor;
};

typedef std::list<Shape> ShapeList;
```

The struct `Shape` then consists of a `Type`, a `Rect`, and two colors. The shape type can be expressed as a named enumeration because any value of type `Type` can have only one of three values: `ELLIPSE`, `RECTANGLE`, or `LINE`. (The integer values are not important.) `Color` is just a `typedef` name for an unsigned long integer. (Using a `typedef` here aids documentation and leaves the design open to later modification.)

For the collection of `Shape` objects, the best solution is probably a list, given that you don't need random access (that is, you will not need to access any shape quickly by position alone.) Lists are easy to append to and economical with memory, and they can be rearranged easily. (That might be crucial later because drawing order is often important.) In any case, you can use a `typedef` to define `ShapeList` as being the same as `std::list<Shape>`. `ShapeList` is easier to type, and it gives you more freedom to change your mind later. You rarely know all the relevant factors early on in a project, and it's wise to keep options open.

There is also a supporting cast of functions and operators to read (and write) rectangles, squares, and other shapes. These are pretty straightforward and exist to make your job easier later. Not only is it easier to type `in >> pt` than `in >> pt.x >> pt.y`, but it means that you can redefine the format for reading in points and know that the rest of the code will be fine. A little more effort up front saves you both time and lines of code later.

Please look at these functions in `chap6\input.cpp`. As in the reverse-Polish program discussed in Chapter 4, you define a `fail()` function that lets you bail out if you hit a problem:

```
void fail(string msg)
{
 throw msg;
}

Point point(double x=0.0, double y=0.0)
{
 Point pt;
 pt.x = x;  pt.y = y;
 return pt;
}

Rect& operator+= (Rect& rt, Point p)
{
  rt.top += p.y;
  rt.right += p.x;
  rt.bottom += p.y;
  rt.left += p.x;
  return rt;
}

void read_color(istream& in, Color& cl)
{
 int r,g,b;
 in >> r >> g >> b;
 cl = TG::rgb(r,g,b);
}

void read_square(istream& in, Rect& rt)
{
    double x,y,rr;
    in >> x >> y >> rr;
    rt.bottom  = y-rr;
    rt.left = x-rr;
    rt.right = x+rr;
    rt.top = y+rr;
}

ostream& operator<<(ostream& out, const Rect& rt)
{
 out << rt.left << ' ' << rt.bottom << ' '
     << rt.right << ' ' << rt.top << endl;
 return out;
}
```

The main input routine is called `read_shapes()`, and it is responsible for reading each line from the file, setting the current foreground and background colors, and including specified plot files.

EXAMPLE

```
bool contains_numbers(const string& s)
{
 string::iterator p1 = s.begin(), p2 = s.end();
 while (p1 != p2) {
   if (isdigit(*p1)) return true;
   ++p1;
 }
 return false;
}

// forward declaration of read_any_shape
void read_any_shape(istream& in, string obj, Shape& shp);

bool read_shapes(string file, ShapeList& sl, real xp, real yp)
{
  Shape shp;
  Color current_bcolor = default_bcolor,
        current_fcolor = default_fcolor;
  string obj;
  Color ival;
 // if there's no file extension, then assume it's .plt   [note 1]
  if (file.find(".")==string::npos) file += ".plt";
  ifstream in;
  if (!in.open(file.c_str())) fail("cannot open " + file);
  while (in >> obj) {
   switch(obj[0]) {
   case 'F': // foreground color
    read_color(in,ival); current_fcolor = ival;
    break;
   case 'B': // background color
    read_color(in,ival); current_bcolor = ival;
    break;
   case 'I': { // insert file!  [note 2]
     string file,rest;
     real xm=xp,ym=yp;
     in >> file;
     // MAY be followed by some (x,y) offset
     getline(in,rest);
     if (contains_numbers(rest)) {
       istringstream ins(rest);
       ins >> xm >> ym;
     }
     // a recursive call!
     read_shapes(file,sl,xm,ym);
  } break;
  default: { // read a shape...[note 3]
    Shape shp;
    read_any_shape(in,obj,shp);
  // shape properties effected by current color and offset
    shp.fcolor = current_fcolor;
    shp.bcolor = current_bcolor;
    shp.rect += point(xp,yp);
    sl.push_back(shp);
  }
 } // switch...
} // while...
  return true;
}
```

The code for the 'I' command (see code marked note 2) calls `read_shapes()` again. This is a classic application of recursion: Plot files can include other plot files, which can themselves include plot files, like Russian dolls or Chinese boxes. The only complication is that you have to check the line after the filename for the **optional** (x,y) offset. The idea is that sometimes the user wishes to bring in a shape file starting at a specific (x,y). The problem is that just saying `in >> file >> xm >> ym` will **always** try to skip to the next line if there are no numbers on the line. So the rest of the input line is read into a string, and `contains_numbers()` is used to check whether it does indeed contain an (x,y) offset. If so, then `istringstream` is used to read that offset in.

If a command is not recognized by `read_shapes()`, then it falls through to the `default` case of the `switch` statement (see code marked note 3). It assumes that the command is a shape and asks `read_a_shape()` to do the specific reading. The resulting `Shape` object is modified by the current color and also by any specified offset. The object is then put into the list of Shape objects.

The function read_a_shape() reads each shape from the file. Because it will be defined later on in the file, it is declared (or *prototyped*) before read_shapes(). It is like a promise that the bill will be paid within 30 days. You can use a simple switch statement here, only looking at the first character of each shape. It would not be difficult to use the whole string, but it might be tedious to have to type the full name of each shape. You can regard squares as degenerate rectangles, and the code of read_a_shape() is straightforward because the detailed work has been done by the input routines such as read_square(). If the command is still not recognized, then fail() is called and an exception is thrown.

Now I could have put the shape-reading code into read_shapes(), but then that function would be a large ugly switch statement, which will only get worse when more commands are added. Plus, putting all the shape commands into their own function makes it easier to see them at a glance.

```
void read_any_shape(istream& in, string obj, Shape& shp)
{
         switch(obj[0]) {
         case 'C': // circle
           shp.type = ELLIPSE;
           read_square(in,shp.rect);
           break;
         case 'E': // ellipse
           shp.type = ELLIPSE;
           in >> shp.rect;
           break;
         case 'S': // square
           shp.type = RECTANGLE;
           read_square(in,shp.rect);
           break;
         case 'R': // rectangle
           shp.type = RECTANGLE;
           in >> shp.rect;
           break;
         case 'L': // line
           shp.type = LINE;
           in >> shp.rect;
           break;
         default:
           fail("Not recognized " + obj);
        }
}
```

The task after reading the shapes is to calculate the bounding rectangle of each of the shapes. You use the standard algorithm `for_each()` to call a function that compares the bounding rectangles (note the initialization of the bounding rectangle because it's easy to get it the wrong way around). Notice that a problem with using `for_each()` here is that you need a global variable to store the overall bounding rectangle because the function is called only with each individual bounding rectangle. You call this `gBnd` to make it obvious that this is a global, and you put it inside a nameless namespace to make `gBnd` (together with `update_bounds()` and `reset_bounds()`) private to this module. Here is the code that calculates the bounding rectangle for all the shapes:

EXAMPLE

```
const double MAX = 1.0e30;
namespace {
  Rect gBnd;

 void update_bounds(const Shape& sh)
 {
  Rect rt = sh.rect;
  gBnd.top    = max((double)gBnd.top,rt.top);
  gBnd.right  = max((double)gBnd.right,rt.right);
  gBnd.bottom = min((double)gBnd.bottom,rt.bottom);
  gBnd.left   = min((double)gBnd.left,rt.left);
 }

void reset_bounds()
{
  gBnd.top = -MAX;
  gBnd.left = MAX;
  gBnd.bottom = MAX;
  gBnd.right = -MAX;
}

} // namespace

void max_rect(const ShapeList& sl, Rect& bounds)
{
  reset_bounds();
  for_each (sl.begin(),sl.end(),update_bounds);
  bounds = gBnd;
}
```

I have included a `write_shapes()` function in `chap6\input.cpp`, which is not necessary for the purpose of drawing shapes. But experience shows that a little extra effort in writing debugging code is always rewarded. Debugging code doesn't have to be polished; it just needs to give a recognizable text form of the program's internal data structures. The `write_shapes()` function, in particular, doesn't write the same format for colors. Instead, it forces the system to output colors as hexadecimal. It is then easy to see the three significant bytes (0xFF is 255, so red is 0xFF0000, blue is 0x00FF00, and green is 0x0000FF). Here is `write_shapes()`, which certainly helped me in constructing this program:

```
void write_shapes(ostream& os, const ShapeList& sl)
{
  ShapeList::iterator sli;
  for (sli =  sl.begin();  sli != sl.end(); ++sli) {
    if (sli->fcolor != default_fcolor)
      os << "F " << (void *)sli->fcolor << endl;
    if (sli->bcolor != default_bcolor)
      os << "B " << (void *)sli->bcolor << endl;
    char ch;
    switch (sli->type) {
    case RECTANGLE: ch = 'R';  break;
    case ELLIPSE:   ch = 'E';  break;
    case LINE:      ch = 'L';  break;
    default:        ch = '?';  break;
    }
    os << ch << ' ' << sli->rect << endl;
  }
}
```

When you finally get to drawing the shapes, you don't need much code at all. With turtle graphics, you use methods for drawing ellipses (`ellipse()`) and rectangles (`rectangle()`) that use the foreground color for the outline, and the background color for the fill color. For instance, to draw a red rectangle with a black border you would set background to red, foreground to black, and then call `rectangle()`. You draw lines by calling `plot()` twice. Here is `draw()`:

EXAMPLE

```
void draw(TG& tg, const ShapeList& sl)
{
  ShapeList::iterator sli;
  for (sli =  sl.begin();  sli != sl.end(); ++sli) {
    tg.set_color(sli->fcolor,true);       // set foreground color
    tg.set_color(sli->bcolor,false);      // set background color
    Rect rt = sli->rect;
    switch(sli->type) {
    case RECTANGLE:
        tg.rectangle(rt.left,rt.bottom,rt.right,rt.top);
        break;
    case ELLIPSE:
        tg.ellipse(rt.left,rt.bottom,rt.right,rt.top);
        break;
    case LINE:
        tg.penup();
        tg.plot(rt.left,rt.top);
        tg.plot(rt.right,rt.bottom);
        break;
    } // switch(...)
  } // for(...)
}
```

This function is the only part of the program where the graphics system and the data come together; to make this program use a different graphics library would be straightforward, because you would only have to rewrite `draw()`.

Extensions

NOTE

You have seen how the basic shape program can be designed and built. I have added some extra more advanced material to show how you could extend this program. Don't be anxious if it seems too difficult at the moment! It does not cover any new material, and you can continue with Part II of this book and come back at any time.

Specifying the coordinates of shapes explicitly is not fun. If you had a graphics processor embedded in your head, perhaps it would seem more natural, but most programmers have to sketch on pieces of paper and then experiment with various numbers. The beauty of true turtle graphics is that everything is done relative to the last position and orientation. For instance, having specified a square, it would be cool to then specify subsequent squares as clones of that square, relative to its position. Here is one possible notation:

```
S 10 20 40
S +w+3
S +w+3
S +h+3 -w-3
```

This is intended to produce three squares in a row, separated by three units. `+w` would mean "add width to the x position" and `+3` would give the extra spacing (which would be optional). The last line will then add a square below the middle square; "+h+3" would mean "add height to y position, plus 3", and so on. I will call these 'relative expressions' since they specify coordinates relative to the current position.

To add this feature, you can modify the code responsible for reading in rectangles (`operator>>`.) First, the code must save the last rectangle read in as a `static` variable `old_rect`. This variable will keep its value between calls to the operator. Second, `operator>>` now scans for ‘w’ and ‘h’ in the line that has been read in, and if the line contains these characters, then it must contain relative expressions . The relative expressions are evaluated with `get_inc_expr()`, which will give us an offset to add to `old_rect`. Otherwise, `rect` is read in as before from the line.

EXAMPLE

```
real get_inc_expr(string s, int idx, int real); // forward

istream& operator>>(istream& in, Rect& rt)
{
 string line;
 static Rect old_rect;
 Point p = point(0,0);
 getline(in,line);
 //------- look for width expr
 int idx = line.find("w");
 if (idx != string::npos)
   p.x = get_inc_expr(line, idx, width(old_rect));
 //------- look for height expr
 idx = line.find("h");
 if (idx != string::npos)
   p.y = get_inc_expr(line, idx, height(old_rect));
 if (p.x != 0 || p.y != 0) { // at least one was specified!
   old_rect += p;            // modify and use last rect...
   rt = old_rect;
   return in;
 }
 // otherwise, we have to read the full rect in
 istringstream ins(line);
 ins >> rt.left >> rt.bottom >> rt.right >> rt.top;
 old_rect = rt;   // save the rectangle for possible future use...
 return in;
}

real get_inc_expr(string s, int idx, real dim)
{
  s += ' ';
  char p_m = s[idx-1];
  if (p_m != '+' && p_m != '-') fail("{+|-}...");
  if (p_m == '-') dim = -dim;
  p_m = s[idx+1];
  if (p_m == '+' || p_m == '-') { // pick up number
    int is = idx+2;
    while (s[is] != ' ') is++;
    string tmp = s.substr(idx+2,is-idx-2);
    real val = atof(tmp.c_str());
    if (p_m == '+') dim += val;  else dim -= val;
  }
  return dim;
}
```

What's Next

In this chapter, you saw some of the more advanced C++ features like function and operator overloading. You saw two non-trivial graphics examples that used recursion.

Up to now, the emphasis has been on functions. The data, as arrays, containers, or `structs`, has been relatively dumb. In the next chapter I am going to introduce another way of looking at programming. In *object-oriented programming*, the data becomes the center of attention, and functions are put **inside the `struct`**. Objects are collections of data that are responsible for managing their own data. You have already been using objects; the standard string, `<iostream>`, and the containers are all examples of the powerful types you can create using C++'s object-oriented features.

Part II

Object Oriented C++

7

Classes

Until now, the focus of this book has been on functions and how to organize the actions of your programs. It is now time to look at organizing programs around data. Object-oriented programming (OOP) considers data to be more important than actions.

In this chapter, you will learn about

- Constructing classes with member functions
- The data access rules for members
- Overloading method operators
- Constructors and destructors
- Separating an interface from an implementation

Member Functions

You have already seen member functions (sometimes called methods) being used. For instance, string has a find() method, which is called using the dot (.) operator. When defining structures, you defined member variables; it is now time to see how to define member functions.

Putting Functions inside Structures

Consider the Point structure from Chapter 5, "Structures and Pointers," which consists of two fields, representing the x and y coordinates of a point. In Chapter 5 we defined the function make_point(), which generates points in a convenient way. A common property of points is their length, defined as the distance from the origin—(0,0)—to the point (x,y). These two points define the corners of a triangle, so the Pythagoras theorem gives the length of this line as the square root of the sum of the sides squares. Instead of making these ordinary functions, you can put them inside a struct, as follows:

```
int sqr(int x)  { return x*x; }

struct Point {
  int x,y;

  void set(int xp, int yp) {
    x = xp;
    y = yp;
  }

  int length()  {
   return sqrt(sqr(x) + sqr(y));
  }

 };

;> Point p;
;> p.set(10,10);
;> p.length();
(int) 14
```

Any function defined inside a struct is called a *member function,* or a *method*. You call a member function by using the dot operator, in the same way you call the methods of standard library objects, such as string. So all struct members—variables or functions—are accessed in the same way: p.x, p.length(), and so on.

Note that you can use any other members within the member function definition without using the dot operator! In the member function `set()`, the member variables `x` and `y` are available directly (as if they were global variables), but of course you could hide them by accident. This is why the preceding example calls the arguments of `set()` `xp` and `yp`. The method `length()` has no explicit arguments: The object is completely implicit.

In most C++ implementations, the object is passed as a hidden reference parameter, so the call `p.length()` is actually equivalent to something like `length(p)`. Otherwise, member functions behave like ordinary functions; they can be overloaded, take default arguments, call themselves recursively, and so on.

The following example adds another function, called `set()`, to the structure shown in the preceding example:

```
struct Point {
  ...
  void set(int val=0) {
    set(val,val);
  }
};
;> p.set(0,0);  // calls the first version of set() with two arguments
;> p.set(0);    // calls the second version of set() with one argument
;> p.set();     // second version, w/ default argument of 0
```

`set()` takes one argument and calls the first `set()` method to do the actual work (this isn't recursion!) The first `set()` is a member of `Point` and so can be accessed directly. Again, we don't need the dot operator since the object is implied.

NOTE

In this example, you could use `x = val; y = val`, but this example shows you how one member function can call another, without using the dot operator.

Why would you want to put functions inside structures? For one thing, it simplifies code because you can simply use `x` instead of `p.x`, and you don't have to explicitly pass the object. But the main reason to put functions inside structures has to do with how we think about data; `length(p)` means "calculate the length of `p`," whereas `p.length()` means "ask `p` for its length." The object `p` becomes responsible for supplying its length to any interested customer, or *client*.

NOTE

With more complicated situations than this example, it is important to have a naming convention. You must be able to tell at a glance whether a variable is a member or just local to a function; I tend to prefix member variables with `m_` (for example, `m_x`, `m_y`), but you can choose any scheme, as long as you're consistent.

Public and Private Members

Object-orientation makes objects responsible for looking after their own data. An analogy is an employee who is delegated a task; the boss does not want to know the details—only that the job is done. Employees need privacy to function; you can't expect them to do a good, responsible job if people keep popping in and replying to their e-mail and writing on their papers. You can mark data and functions as being private within a `struct`. Here is a simplified definition of a `Person` `struct`:

EXAMPLE

```
struct Person {
  private:
    string m_name;
    long   m_phone;
    string m_address;
    long   m_id;
  public:
    void set(string name, string address, long phone, long id) {
       m_name = name;   m_address = address;
       m_phone = phone; m_id = id;
    }

    string get_name()     { return m_name; }
    string get_address()  { return m_address; }
    long   get_phone()    { return m_phone;  }
    void   set_phone(long ph) { m_phone = ph; }
    long   get_id()       { return m_id;     }
 };
;> Person eddie;
;> eddie.set("Eddie","",6481212,554433);
;> eddie.get_name();
(string) 'Eddie'
;> eddie.m_name;
CON 11: Cannot access 'm_name'
(string) 'Eddie'
```

The `Person` object has some privacy to manage its own data. Any code outside the structure cannot access the member variables directly. My attempt to directly access `eddie.m_name` produced an error message, since `m_name` is declared private to `Person`. To modify `m_name`, outside code has to go through the proper channels and use the *accessor functions* such as `get_name()`. Accessor functions are often called *get* and *set methods*.

Notice that all members after `private:` will be private to the `struct`, until a `public:` statement is encountered. In Java, each member has to be explicitly marked public or private.

`struct` and `class`

At this point, we will begin to use the keyword `class` instead of the keyword `struct`. The resulting type is exactly the same; the only serious difference between `struct` and `class` is that classes are *private* by default and you always have to use `public:` to make your members visible to the rest of the world. Generally, member variables are kept private, and you define get and set methods for public access.

This seems at first to be both fussy and inefficient, but C++ does not penalize you for this programming style. When you write methods inside the class body (that is, everything in the braces following `class`), the compiler tries to *inline* them. That is, the machine code is directly injected in place of a function call. If the function is a simple one, such as a member variable access function (for example, `get_id()` in the `Person` class above), then this is just as efficient as accessing the variable directly. That is, `eddie.get_id()` is likely to be just as fast as `eddie.m_id`. In general, you should not worry about the extra cost of making a function call until you know for sure that the cost is unacceptable.

The Idea of Encapsulation

As you learned in the preceding section, C++ does not penalize you for separating data from the interface. If an object is going to manage its own data, then this separation is important. In the object-oriented view of things, accessing a class's data directly is bad manners, because it makes it hard for the class to take full responsibility for the data. This principle, that data should be private, is called *encapsulation*. In the next few sections, I want to show you why this is a good principle to follow.

Protecting the Representation

It is possible to manage complex organizations only if the various tasks are delegated properly. People are given tasks and responsibility for finishing them; they should not bother their boss with details, and they should decide precisely how to do what they need to do. This human-management analogy is useful when we're talking about programming because large software systems are the most complicated things ever put together by human beings, and they depend on cooperation between their subsystems. Traditional programming often suffers from micromanagement because roles and responsibilities are not clearly defined. The boss tends to get too involved and spread too thin, and the employees never mature.

The idea of an employee is very close to the idea of supervision, so perhaps it's best to think of objects as contractors; they are given the specification and have to deliver the goods. For instance, if you are writing a C++ program, you don't want to know the details of each `iostream` and `string` object; the system contracts out the job of managing strings to the `string` class. Very occasionally, standard strings are too expensive for a particular job, and then you either do it yourself or find another `string` class to handle it (that is, you get another contractor). If you use a `string` class, then you should not have to know how strings are represented because managing the data is the responsibility of the `string` class. Furthermore, if you start fooling around with the representation, micromanaging the project, the class cannot do its job properly.

Is it always necessary to keep data private? People have strong feelings about this question. Some "pure" object-oriented languages such as Smalltalk never expose their objects' data; even numbers are objects. Smalltalk is therefore incredibly slow. Stanley Lippman (see his excellent *C++ Primer*, 3rd edition, Addison-Wesley) came up with a useful rule of thumb: For simple things, such as a geometric point, the *interface* is the same as the *representation*. So there is nothing wrong with accessing the member variable `x` of `Point` directly, and saying `p.x`.

The `Date` Class

You have seen dates handled several ways in this book, and so a `Date` class would be appropriate (not to mention traditional). In this section we will create a `Date` class that has an odd representation.

Most commercial data processing was done in the past (and is often still done) with COBOL, which keeps data in records. These records work rather like C/C++ `structs`, except that numbers are traditionally stored as 'pictures' (that is, as characters). So a date would be represented with six characters, such as `590320`. Because memory and storage in the Sixties was very expensive (more expensive than programmers), programmers used two-digit year fields. Another reason was that everything had to fit onto 80-character wide punch cards. The following is a `Date` class built around a six-character representation:

```
class Date {
  char m_data[6];

  int read2(int i)
  {
    char tmp[3];
    tmp[0] = m_data[i];
    tmp[1] = m_data[i+1];
    tmp[2] = '\0';
    return atoi(tmp);
  }

  void write2(int i, int val)
  {
    char tmp[15];
    itoa(val,tmp,10);
    if (val < 10) {
      m_data[i] = '0';
      m_data[i+1] = tmp[0];
    } else {
      m_data[i] = tmp[0];
      m_data[i+1] = tmp[1];
    }
  }

public:

  int  year()
  {
    return read2(0) + 1900;
  }

  void year(int y)
  {
     write2(0,y - 1900);
  }

  int  month()      { return read2(2); }
  void month(int m) { write2(2,m); }

  int  day()        { return read2(4); }
  void day(int m)   { write2(4,m); }
};
```

The business end of this class (that is, the public interface) consists of get/set methods. Rather than calling them `get_year()` and `set_year()`, this example uses overloading to use the method named `year()` for both the get and the set methods. (The two methods have such different signatures that it is unlikely that anyone would get them confused, and using just one short name avoids unnecessary typing.) The get version of `year()`, `month()`, and `day()` relies on the private method `read2()` to read two characters and convert it to an integer; in the string '590320' the year is at offset 0, the month is at offset 2, and the day is at offset 4. In turn, `read2()` works by copying two characters into a buffer, making sure that there's a `NULL` character at the end, and using the C library function `atoi()` ("**A**SCII **to** **I**nteger") to convert this to an integer.

Similarly, the set methods use `write2()`, which uses the library function `itoa()` ("**I**nteger **to A**SCII") to generate a character representation of an integer. It can be difficult to remember how to use functions like `itoa()`, so this example class separates out (or *factors out*) the confusing integer conversion code into the separate method `write2()`. There is also a special case when the day or month is less than 10 (for instance, 9 must be written out as '09').

Member functions, like `write2()`, which are part of the implementation, are usually kept private and are often called *helper functions*. Factoring out common code can save you a lot of cut-and-paste programming and will result in cleaner code. A good object-oriented programmer needs to remember to be a good structured programmer on occasion.

Dealing with the Year 2000 Problem

Everybody got a little overexcited about the year 2000 problem, but the problem really boiled down to lack of encapsulation. The odd (and non-efficient) `Date` class would not have caused a problem because it isolated the six-character representation issue in one place only. All client code of `Date` would be religiously calling `Date::year()`, and making the data private would ensure that no one got careless. It is, therefore, not necessary to change each and every reference to the year throughout all of the code.

There would then be two kinds of Year 2000 fixes. The first kind is necessary when people don't want to mess up their existing file formats; in this case, dates must continue to be represented by six digits. This can be done by choosing a *pivot* value for the date; you assume, for example, that the years 00 to 99 represent 1930 to 2030. You replace the number 1900 within the preceding code with the number 1930, you recompile everything, and the fix is made. You still have to patch the existing files and hope that no pre-1930 dates are included in the file.

There are, however, a fair number of people older than 70 years, (that is, born before 1930) so an insurance company, among others, would not find this fix acceptable. So in the second kind of Year 2000 fix, you need to redo the old file format and rethink the representation. A date could be packed into four binary bytes; two bytes for the year (measured from some convenient point) and one byte each for the month and the day. This example shows another `Date` class, which is in fact simpler than the first one:

EXAMPLE

```
class Date {
  short m_year;
  unsigned char m_month, m_day;
public:
  int  year()      { return m_year + 1600; }
  void year(int y) { m_year = y - 1600;    }
  int  month()     { return m_month;       }
  void month(int m){ m_month = m;          }
  int  day()       { return m_day;         }
  void day(int d)  { m_day = d;            }
};
```

This solution is faster than the original class, and it's more space-efficient as well. Because this `Date` has exactly the same set of public members as the old `Date`, it is possible to use it as a replacement. In other words, they both support the same *interface*. If done properly, object-oriented programming makes it possible to build real software components, which can be replaced just like electronic components. A standard serial mouse on a computer can be replaced by another standard serial mouse without any reconfiguring, because both devices support the same interface.

Of course, a real `Date` class would have checking code. This is another very good reason for keeping the data private and separate; if everyone has to go through the same gate to modify the date, then you can always catch attempts to set a nonexistent date (such as February 31, 2001). You may then take action, like throwing an exception. Catching errors as early as possible is the best way to make programs robust.

NOTE

Privacy does not mean *secrecy*. You let objects maintain privacy because it is essential to their jobs, not because you are trying to hide details from other programmers. (The C++ mechanisms for hiding details from others—when it becomes necessary, as with commercial libraries—are using separate compilation and dynamic link libraries.) It is clear from the class definition what members are private; this tells clients of that object that they don't have to worry about the details but just use this public interface. This prevents the client code from taking shortcuts, like a picket fence stops intruders: It isn't a physical barrier, but it defines a boundary.

`const` Methods, the `this` Pointer, and Static Methods

Objects can be passed by reference, and these references can be marked as `const`, which means they should not be modified. But how do you do this with methods, where the object argument is hidden? As mentioned previously, the usual strategy is to pass the object as a hidden reference argument. Sometimes you need to access this argument (for instance, if a method of `Point` had to call a function that took a `Point` argument). The `this` pointer is always available, and its value is a pointer to the object. For example, within methods of `Point`, `this` has type `Point*`. (The following code assumes that `operator<<` has been overloaded for `Point`.)

EXAMPLE

```
struct Point {
  ...
  void show() {
    cout << *this << endl;
    this->x = 1;  // silly, but legal
    this->y = 2;
  }

  Point& translate(const Point& p) {
    x += p.x;  y += p.y;
    return *this;
  }
};
;> Point p1,p2;
;> p1.set(120,300);  p2.set(30,50);
;> p1.show()
(120,300)
;> p1.translate(p2).show();
(150,350)
```

The `this` pointer is usually used as `*this`, which in the preceding example has type `Point&`; it is not a variable, but it usually isn't a `const` reference. Notice that the return type of `translate()` is a reference to a `Point`, which can be used by any other method; this is similar to the technique that makes overloading `operator<<` so useful, where you can build up a chain of function calls.

The method `show()` is both silly and dangerous; it is silly because `this->x` is spelled out in full (although in some respectable code the authors do this to indicate when they're accessing a member field), and it is dangerous because a user would probably not expect a method called `show()` to actually modify the object. In the same way, if I saw a method called `get_x()` I would be very surprised if it modified its object. There are no language rules against playing this kind of trick on users, but at least you should be able to insist that the `this` pointer is a `const` reference. You do this by using `const` after the method declaration, which serves two purposes: It means you (the writer of the method) cannot accidentally change the object, and the client (the user of the method) has a guarantee that the object will not be modified. As with passing `const` references, you should get into a habit of labeling methods as `const`, unless they actually modify the object. Here `show()` has been redeclared as `void show() const`; it will now be an error to modify `x` or `y` in `show()`:

```
struct Point {
  ...
  void show() const {
    cout << *this << endl;
   // would be an error now to modify x or y
   // this->x = 1;
   // y = 2;
   }

 void show_x() {
   cout << "x = " << x << endl;
 }
};

void show_point(const Point& p)
{   p.show();   }       // fine!

;> void call_p(const Point& p)
;> {  p.show_x();  }       // an error!
CON 20:cannot call a non-const method with a const object
```

Please note that if you do have a `const` reference to a `Point`, then you cannot call a non-`const` method like `show_x()`. This is because `show_x()` cannot guarantee to `call_p()` that it will not modify the object `p`. This is considered an error by most C++ compilers (although I note that BCC32 still only gives a warning). The GCC error message is a little cryptic at first:

```
C:\bolsh\ucw\examples>c++ -c cpoint.cpp
cpoint.cpp: In function `void call_p(const Point &)':
cpoint.cpp:10: passing `const Point' as `this' argument of `void Point::show_x()'
➥discards qualifiers
```

It is really trying to tell you that `const Point& p` cannot be converted to `Point& p`, because the qualifier `const` would have to be dropped.

Not all member functions have a `this` pointer. For example, you can declare a method to be `static`, and it is then a plain member function. It remains a privileged member of the class, but it does not operate on any particular instance of that class. Here is an example of a `Point` that has a `static` member function `make()`. `make()` has access to the private members of any `Point` object but has no `this` pointer. Notice that static functions are called just like members of a namespace, using the scope operator (`::`)—in fact, you can think of a namespace as a class that has only static members.

```
;> class Point {
;1}  int x,y;
;1}
;1} void set(int X, int Y) { x = X; y = Y; }
;1} static Point make(int X, int Y) {
;2}   Point p;
;2}   p.set(X,Y);
;2}   return p;
;2} }
;1} };
;> Point p;
;> p = Point::make(200,100);
(Point&) Point {}
;> p.x; p.y;
(int) 200
(int) 100
```

Constructors and Destructors

An object will often need to be put into some initial state. The `set()` method does this for `Point` objects, as does `make()`. This does fine for simple classes like `Point`, but more complex classes need to set themselves up properly for business. You cannot safely use such an object without initializing it, and it is easy to forget to explicitly initialize an object. Constructors are special methods that will automatically be called when an object is declared.

Objects often allocate memory dynamically and, therefore, require destruction as well as construction: Any pointer allocated with `new` must be given back to the system with `delete`, etc. Destructors are methods that will be called when the life of an object is over.

Class Constructors

Local variables will often not contain sensible values after declaration (unless they were explicitly initialized, of course). This is because non-static locals are genuinely temporary; some space is made on the stack for them, and when the function ends, that space is taken back by popping the stack. That space can then be used for somebody else's local variables, so local variables will contain whatever the last stack value was. Unlike Java, C++ does not implicitly initialize ordinary variables; people may not need it, and the motto of C++ is "you never pay for what you don't use." However, you can specify code that can be automatically called whenever an object is created. These special functions are called *constructors.* A class constructor has the same name as the class (or `struct`), and no return type; otherwise, it behaves like any other method. Here is a constructor for the `Point` example (the full definition of `Point` is available from `chap7\points.h`):

```
struct Point {
  int x,y;
  void set(int xp, int yp) {
      x = xp;  y = yp;
  }

  Point(int xp, int yp) {
     set(xp,yp);
  }
};
;> Point p(100,120);
;> p.x; p.y;
(int) 100
(int) 120
```

Previously, we defined non-member functions such as `make_point()` or member functions like `set()` to do the important job of initializing an object to a known value, but constructors are easier to use and often more efficient (because `make_point()` relies on copying the structure). Constructors are also called when an object is created dynamically, as in the following example:

```
;> Point *ppoint = new Point(40,23);
;> ppoint->x;
(int) 40
```

Default Constructors

With the constructor shown in the preceding section, you cannot declare a `Point` without supplying initial values. The error message is instructive: The compiler tells you that no constructor of `Point` can match a signature that has no arguments. Here is what happens:

EXAMPLE

```
;> Point pp;
CON 4:Function cannot match 0 parameters
CON 4:no default constructor
;> Point p();
;> p.x;
CON 12:Not a class object or pointer
```

Curiously, you can get away with using `Point p()`, but the result is not what you expect. The compiler considers this to be a forward declaration of the **function** `p()`, which returns a `Point` value and takes no arguments—hence the error message. Watch out for this because it might seem logical to declare a `Point` like this, but C++ (like English) is not necessarily logical.

The constructor with no arguments is called the *default constructor,* and it is used when the object is declared without arguments. Constructors, like any other functions, can be overloaded, so you can add another constructor, this time taking no arguments:

```
struct Point {
  ...// same as before
  Point() {
     set(0,0);
  }
};
```

It is now possible to declare `Point p`, which results in a `Point` object that is always properly initialized.

A class can contain objects that need construction. If you supply a constructor, C++ guarantees that the objects in the class will also be constructed. If you do not supply a constructor, the compiler will generate a default constructor that will do this.

The question now is "How could the original definition of `Point` have worked?" What happens for plain old structures? If you do not supply a constructor for a plain structure, C++ assumes that there is no need for construction, and it doesn't bother to find the default constructor. If the class contains objects (such as strings) that need construction, the compiler generates a default constructor, which guarantees that the objects are constructed. Unlike other object-oriented languages, C++ is usually pretty good at handling that kind of detail for you. For example, an (apparently) simple class that contains some standard `string` objects automatically acquires a default constructor. Any `string` objects will be initialized to the empty string (`""`). You need to be aware of this because occasionally you will forget to supply a default constructor, and a straightforward class will refuse to compile.

Why doesn't the system supply a default constructor automatically for `Point`? As soon as you define any constructor, C++ assumes that you want to take full responsibility for constructing that object. It assumes that you are doing some special initialization, and it doesn't want to second-guess your needs.

Explicit Clean-up

Objects often grab system resources, such as memory or file handles. A tiresome task in traditional programming is remembering to give these resources back to the system. If we didn't take our library books back, eventually the library would run out of books. (Java has a different philosophy, called *garbage collection*: Occasionally the librarians let themselves into your house and pick up your library books.) If you allocate memory with the operator `new`, at some point you must use `delete`; if you open a file for writing, then you must close it. Failure to close files might mean that data isn't written to the file; with the `iostream` library, you are writing to a buffer that is *flushed* to the file when it's properly closed. (Windows puts some further buffering between the file system and the actual disk.) But I have told you not to worry about explicit closing of files. How is this file-closing achieved automatically?

You can define *class destructors*, which are designed to clean up and "unconstruct" objects. Destructors do not have return types, the same as constructors, and they use the tilde symbol (~). For example, this is (roughly) how `ofstream` closes a file:

```
class ofstream ... {
 ...
 ~ofstream() {
    close();
  }
};
```

A class's destructor is very rarely called directly. Usually it is automatically called at the end of a variable's career. If an `ofstream` object is declared in a function, the destructor guarantees that the file will be closed, however the function ends. You can jump out of the function by using `return` at any point, or if an exception is thrown; in either case the destructor will be called.

For example, you can add to the `Point` class a destructor that does nothing but announce the object's destruction. The following code then defines a function `test_destructor()` that shows the three main ways of leaving a function: returning from some arbitrary point, leaving when an exception terminates the function's execution, and exiting normally. `test_destructor()` is exercised by `catch_it()`, which will catch any exceptions thrown:

EXAMPLE

```
struct Point {
  ...
  ~Point() {
    cout << "Destroyed " << x << ' ' << y << endl;
   }
 };

 void test_destructor(int how) {
   Point p;
   switch(how) {
   case 0: return;
   case 1: throw "exception";
   default: break;
   }
   cout << "normal exit\n";
 }

 void catch_it() {
    try {
      test_destructor(1);
    } catch(...) {
      cout << "caught!\n";
    }
 }

;> test_destructor(2);
normal exit
Destroyed 0 0
;> test_destructor(0);
Destroyed 0 0
;> catch_it();
Destroyed 0 0
caught!
```

Just as C++ guarantees that members of a class (or struct) will be properly constructed, it guarantees that they will be destroyed if they have destructors. If necessary, the compiler can generate a destructor. In the next example, you define the structure `Line` as consisting of a start and an endpoint, which are both of type `Point`. Rather than defining a function, you can put declarations within a block when in interactive UnderC mode. The `Line` object `lin` is a local variable that goes out of scope after the close brace; you then see that the points were properly destroyed. That is,

```
struct Line {
  Point start;
  Point end;
};
;> { Line lin; }
Destroyed 0 0
Destroyed 0 0
```

The `delete` operator also causes the destructor to be called. In this case, the programmer completely controls the lifetime of the object:

```
;> Point *pp = new Point(3,4);
;> delete pp;
Destroyed 3 4
```

Operators as Methods

C++ operators are functions, and like all functions they can be members of a class. In Chapter 6, “Overloading Functions and Operators” you saw how operators can be redefined for user-defined types. There are some operators that can be defined only as class members, however.

The [] Operator

We previously overloaded `operator+=` for `Point`; here is how it looks as a method of `Point`:

```
struct Point {
  ...
   Point& operator+= (const Point& p) {
     x += p.x;  y += p.y;
     return *this;
   }
 };
;> Point p(20,30), offs(10,10);
;> p += offs;
(Point) p = Point {}
;> p.x; p.y;
(int) 30
(int) 40
```

The `+=` operator is, in fact, just another name for the translate operator—that is, it moves the point along. The non-member version has two arguments, which here is implicit as the `this` pointer, but both versions are used the same way. In fact, you must either define `operator+=` as a member or a non-member.

Some operators can be defined only as members. For instance, the operator `=` can be only a method because an object must have full control of assignment (and any other copy operations). Allowing client code could redefine what assignment means and would lead to confusion.

`The` operator `[]` is interesting. Array access is considered an operator, and you can redefine it. Here is a very simple `Array` class; it isn't resizable, but (unlike regular arrays) it is range-checked:

EXAMPLE

```
const int N = 100;

class Array {
   int m_arr[N];
 public:
   int& operator[] (int i)
   {
     if (i < 0 || i > N) {
       cerr << i << ": array index out of bounds\n";
       throw 0;
     }
```

```
      return m_arr[i];
    }
  };
;> Array a;
;> a[10] = 2;
(int&) 2
;> a[10];
(int&) 2
;> a[1021];
1021: array index out of bounds
uncaught exception: 11 array.h
```

Note that `a[index]` can be used on both the right- and the left-hand sides of an assignment. `Array::operator[]` returns a reference to an integer; assigning to this `reference` causes the value to be updated.

Using Methods versus Functions

C++ can be confusing because there is often more than one way to do a particular thing. For instance, we can define `operator+=` as either a member or a nonmember operator. In the case of `operator+=`, the member form is both simpler and more appropriate than the nonmember form because `+=` is an assignment operator.

In the past, people overused methods because there was no other way to create a separate namespace. Explicit namespaces now offer an alternative to defining everything as a method. Methods should represent the basic operations that need raw access to the data. In particular, you shouldn't add very specialized operations that are useful only in some applications. Instead, you should be able to define functions (in some namespace) to do the job. (Alternatively, as will be discussed in Chapter 8, "Inheritance and Virtual Methods," you can use inheritance, which is a better way to specialize a class for a particular job.)

A useful way of looking at classes is the idea of *abstract data types* (ADTs). An ADT is defined by a set of operations (called the *interface*), rather than by representation. For instance, in mathematics integers are defined by addition, subtraction, multiplication, and division. Some operations, such as the remainder (modulo) operation, can be defined in terms of these functions, but the remainder operation is generally useful and so it's made into an operator. Counting the number of digits in a decimal representation, on the other hand, is not a basic operation and would be made a function.

For instance, consider the idea of a stack which you met in Chapter 3, in the section "Stacks and Queues." Considered as an ADT, a stack is defined by the operations `push()`, `pop()`, and `depth()`; it's also convenient to have empty(), which is the same as depth()==0. Here is an implementation of a Stack class, using a list:

```
class Stack {
private:
   list<int> m_list;
public:
   void push(int i)    // push a value onto the stack
   {
    m_list.push_back(i);
   }
   int pop ()         //  pop a value from the stack
   {
     int val = m_list.back();
     m_list.pop_back();
     return val;
   }
   int depth() const  //  how deep is the stack?
   {
     return m_list.size();
   }
   bool empty() const //  is the stack empty?
   {
     return depth()==0;
   }
};
;> Stack s;
;> s.push(20);
;> s.push(30);
;> s.depth();
(int) 2
;> s.pop();
(int) 30
;> s.depth();
(int) 1
;> s.pop();
(int) 20
```

It is interesting to compare this to the version of this example in Chapter 4, “Programs and Libraries,” which uses an array. That previous version defined a stack within a namespace. The advantage of using a class is that you can have a number of stacks. The advantage of using a list is that it can grow as the stack grows, without using more memory than necessary; the disadvantage is that it may be too slow for mission-critical tasks, such as the UnderC stack machine.

Interfaces and Implementations

However you write a stack class, the interface of a stack is defined by the operations `push()`, `pop()`, and `depth()`. These methods define the interface, and the actual code and data is called the implementation. It is very useful to separate these two different aspects of a class. A separate interface is easier to understand, and the implementation can be separately compiled.

Creating a Header File

To support separate compilation and to break a program into self-contained units, C++ encourages the use of header files. In the case of a function, a header file would contain prototypes, which would then be fully defined in another file. The equivalent for a class is similar. Within the class body are the member declarations, for both the data and the functions. Any client code that wants to use the `Stack` class only needs to include this header file:

```
// stack.h
#include <list>
class Stack {
private:
   std::list<int> m_list;
public:
   void push(int i);
   int pop ();
   int depth() const;
   bool empty() const;
};
```

This header file defines the interface to the class, and the implementation of the class is then defined elsewhere. The user of this class might not have the source code for the implementation; although having the source code can be useful, it is not essential. All the crucial information is present in the class definition and comments.

You should now be able to see how useful it is to label methods as `const`. When you do, the user does not have to refer to the implementation to know that `Stack::depth()` does not change the object. Note that the interface does not completely hide the details of the implementation; the user can make a pretty good guess that the stack uses a `list`. As previously discussed in Chapter 4, "Programs and Libraries," in the section "The `std` Namespace," you make it explicit that you are using the standard `list`, rather than just taking namespace `std` for granted.

The Implementation File

The implementation file for `Stack` looks like this, with the scope operator (`::`) used to show that the functions are members of the `Stack` class:

EXAMPLE

```
// stack.cpp
#include "stack.h"
 void Stack::push(int i)
 {
   m_list.push_back(i);
 }
 int Stack::pop ()
 {
    int val = m_list.back();
    m_list.pop_back();
    return val;
 }
 int Stack::depth() const
 {
    return m_list.size();
 }
 bool Stack::empty() const
 {
    return depth()==0;
 }
```

Note that you must have previously declared these methods in `stack.h` before you can define them like this; otherwise, the compiler complains that the method is not a member of the class. (Incidentally, you can define the members of a namespace in exactly the same fashion, and it is, in fact, the recommended way.) Must all methods of a class be separately defined like this? Simple set and get methods can be left in the header file, where they will are automatically inlined, as discussed in the section "`struct` and `class`."

Separating the Interface from the Implementation

Separating the interface from the implementation of a class allows you to organize a program into well-defined modules. Practically, it is useful because it means that rebuilding a million-line program does not require compiling a million lines of code; usually only the implementation of a class changes, not the interface. You should not have to recompile a whole program just because of a small change in one method of `Stack`. This is the advantage of keeping even simple get/set methods out of the interface.

From an organizational point of view, separating the interface from the implementation means that responsibilities are kept separate as well. For example, you can think of the `iostream` library as the packaging and logistics department of a C++ program; it is concerned with how to physically get output out onto the screen, or into a file, or whatever. The client code merely has to ask.

Also, a programmer should not be swamped with details. Keeping the implementation separate is often called *information hiding* and goes hand-in-hand with encapsulation.

Case Study: Wrapping a Class for Downloading Web Pages

An important use of classes is that they can provide a convenient wrapper around an existing Application Programming Interface (API). APIs in Windows (as in most operating systems) are collections of functions that must be called in a particular sequence. When the API is thus wrapped up as a class, such a class becomes independent of the precise details of that API and can be ported with little fuss to some other platform. Also, as you will see, the resulting class is often much easier to use than the original code.

Using the WinInet API to Access Web Pages

The example in this case study is the WinInet Win32 API, which is a set of high-level functions that implement the raw business of WinSock calls and Internet protocols. It is much simpler than doing it yourself. You may only have a dial-up connection, but even so, it's easy to set up a little Web server such as HSWeb (`www.heat-on.com`) to operate locally on your machine. The IP address `127.0.0.1` is reserved for such connections.

However, with WinInet, as is often the case with APIs, there are dozens of options, and code using this API can be less than dazzlingly obvious. If the code seems strange at first, you are in good company. The API involves straightforward function calls, but each function takes a lot of arguments, most of which you do not have to worry about.

It helps to understand some basic concepts: Accessing a Web page with WinInet involves opening an Internet *session*, then opening a *connection* to the server, and creating a *request*. Finally, the request is sent to the server and the data is retrieved. Everything in Win32 API programming is accessed with *handles*, which are indirect references. Handles are a form of information hiding; Win32 function calls very rarely allow you to actually modify the system's data, but only to refer to it using handles.

All the WinInet functions in the following example begin with `Internet` or `Http`:

EXAMPLE

```
// wininet1.cpp
#include <windows.h>
#include <wininet.h>
#include <iostream>
using namespace std;

HINTERNET hSession,hConnection,hRequest;

bool go_get(char *hostname, char *page, char *buffer,
            long n_to_read)
{
 hSession = InternetOpen("wininet1",
                INTERNET_OPEN_TYPE_DIRECT,
                NULL,NULL,NULL);

 hConnection = InternetConnect(hSession,
                 hostname,INTERNET_DEFAULT_HTTP_PORT,
               NULL,NULL,
               INTERNET_SERVICE_HTTP, NULL,0);

 hRequest  = HttpOpenRequest(hConnection,
             "GET",page,
             NULL,NULL,NULL,
            INTERNET_FLAG_RELOAD |
             INTERNET_FLAG_EXISTING_CONNECT,
            0);


 if (hRequest==NULL) {
   InternetCloseHandle(hSession);
   return false;
 }

unsigned long dw1,dw2, n_actually_read;

 bool send_req  = HttpSendRequest(hRequest,NULL,0,NULL,0);
 bool has_read  = InternetReadFile(hRequest,
                    buffer,n_to_read, &n_actually_read);
 cout << "read: " << (int)n_actually_read << endl;
 InternetCloseHandle(hSession);
 return has_read;
}
const int MAXSIZE = 124096;
char buff[MAXSIZE];

int main()
{
  if (go_get("127.0.0.1","index.htm",buff,MAXSIZE))
    cout << buffer << endl;
}
```

```
c:\ucw\chapter7> c++ wininet1.cpp -lwininet
```

In this example, an Internet session is opened with `InternetOpen()`. With this handle, a connection to the actual server is opened by using `InternetConnect()` and the server's hostname (for example, `www.bonzo.com`). A "GET" request for the page (for example, `/default.htm`) must be created with `HttpOpenRequest()`, and this request is then sent with `HttpSendRequest()`. Finally, the resulting output returned by the server is read by `InternetReadFile()`.

All of these functions return either `true` or `false`. A real program would check these return values and clean-up afterwards. This involves closing the session handle, which automatically closes the connection.

The last line shows how to compile this program. The `-lwininet` tells the linker to import the WinInet functions from Windows. Appendix D, "Compiling C++ Programs and DLLs with GCC and BCC32" will tell you more about this process.

There are dozens of parameters and flags in this example code; they control whether you go through a proxy server (`INTERNET_OPEN_TYPE_DIRECT`), whether you grab from the cache (`INTERNET_FLAG_RELOAD`), and so on. (It's no wonder that people give up on C++ and go do Perl!) You need to include many lines from header files to compile the preceding code, which is one of the irritating things about Windows API programming. For instance, in GCC you get about 20,000 lines included; using the Microsoft compiler CL you get more than 300,000 lines included! There must be an easier way to do this, and indeed there is.

Encapsulating WinInet

The following is the definition of a class that encapsulates the functionality we've been talking about:

EXAMPLE

```
// inet++.h
#include <string>
const int n_to_read = 4096;
```

```
#ifndef _IMPL_
typedef void *HINTERNET;
#endif

class HTTP {
private:
  typedef std::string String;
  HINTERNET hSession,hConnection,hRequest;
  char buffer[n_to_read];
 public:
   HTTP(bool using_proxy=true, char *name = "wininet");
   void close();
   ~HTTP();
   bool      available()  { return hSession != NULL; }
   HINTERNET request()    { return hRequest; }

   void open(String hostname);
   void create_request(String page, bool do_get);
   void send_request(String data);
   int read(char *buffer, int nz);
   int read_file(String page, String filename, string data);
   int get(String page, String filename = "");
   int put(String page, String data, String filename);
};
```

`#ifndef` is a *conditional compilation*. If the macro `_IMPL_` is defined, then `HINTERNET` is defined as a simple `(void *)` pointer. That is, you don't have to include all those Windows headers just to define one handle type. This is a case where information hiding is a practical way to reduce compilation times. Besides, the idea is to keep all the operating system–specific code wrapped up neatly in a few modules.

As you can see in the implementation file, the macro is defined before the header is included:

NOTE

Most of the code in this example is the same as the code in the preceding example, and the purpose is never to have to write this API code again.

EXAMPLE

```
#define _IMPL_
#include "inet++.h"

HTTP::HTTP(bool using_proxy, char *name)
{
    hSession = InternetOpen(name,
                 (! using_proxy ? INTERNET_OPEN_TYPE_DIRECT :
                   INTERNET_OPEN_TYPE_PRECONFIG)
                   NULL,NULL,NULL);
    hRequest = NULL;
    hConnection = NULL;
}

void HTTP::close()
{
    InternetCloseHandle(hSession);
}

HTTP::~HTTP()
{
    close();
}

void HTTP::open(string hostname)
{
  if (hSession == NULL) fail("internet open failed");
  hConnection = InternetConnect(hSession, hostname.c_str(),
                  INTERNET_DEFAULT_HTTP_PORT,
                  NULL,NULL,
                  INTERNET_SERVICE_HTTP, NULL,0);
  if (hConnection == NULL) fail("connection failed");
}

void HTTP::create_request(string page, bool do_get)
{
   hRequest = HttpOpenRequest(hConnection,
                (do_get ? "GET" : "POST"),
                page.c_str(),
                NULL,NULL,NULL,
                INTERNET_FLAG_RELOAD | INTERNET_FLAG_EXISTING_CONNECT,
            0);

   if (hRequest==NULL) fail("request failed");
 }

 void HTTP::send_request(string data)
 {
   const char *str = NULL;
   if (data != "") str = data.c_str();
   OUT(str);
   if (!HttpSendRequest(hRequest,NULL,0,
     (char *)data.c_str(),data.size()))  fail("send request failed");
 }
```

```
int HTTP::read(char *buffer, int nz)
{
   DWORD n_actually_read;
   if (!InternetReadFile(hRequest,buffer,nz, &n_actually_read))
        fail("read file failed");
  // necessary to put a final NUL character!
   buffer[n_actually_read] = '\0';
   return n_actually_read;
  }

 int HTTP::read_file(string page, string filename, string data)
 {
 // note: no data means a plain GET...
   create_request(page,data == "");
   send_request(data);
   int n = read(buffer, n_to_read);
   buffer[n] = '\0';
   // writing out in ASCII mode should sort out any UNIXisms.
   ofstream out(filename.c_str());
   out << buffer << endl;
   return n;
 }

 int HTTP::get(string page, string filename = "")
 {
   if (filename == "") filename = strip_filename(page);
   return read_file(page,filename,"");
 }

 int HTTP::put(string page, string data, string filename)
 {
   return read_file(page,filename,data);
 }
```

The constructor `HTTP()` has two arguments: if the argument `using_proxy` is `true` (which is the default), WinInet tries to use the Windows settings. So the default for this class should work, whether there is a proxy server or not; the flag is for occasions when you definitely don't want to go through the proxy, such as with a local Web server.

The business end of this class includes two methods: `get()` and `put()`. By default, `get()` strips the filename out of the full page and creates a local file of that name (so `/~ldubb/friends/pia.html` is saved as `pia.html` in the current directory).

You can use the method `put()` to submit forms to a Web server. For instance, `dog=bonzo&cat=felix%20the%20second` would be a `POST` request from a form that contains two edit boxes, `dog` and `cat`; any nonalphabetical characters, such as spaces, are encoded as their hex value (`%20 = 32 = ' '`). The resulting output is retrieved into the filename.

You don't need to be an expert on WinInet or Internet protocols in general to know that if you need finer control over these API calls, you can expand the class `HTTP`'s interface. This example uses the default server port for Hypertext Transfer Protocol (HTTP). It would be easy to make `m_server_port` a member variable, defaulting to `INTERNET_DEFAULT_HTTP_PORT`, which could be passed to the constructor or set with a method.

This code makes extensive use of `fail(msg)`, which means (as mentioned in Chapter 4) "throw `msg`." If an exception is thrown, you can exit the `try` block cleanly, calling HTTP's destructor in the process. This guarantees that all open handles are closed.

The following program shows `inet++` in action; it certainly is a good deal easier to write (and maintain) than straight WinInet API code:

```
// test-inet.cpp
#include "inet++.h"
int main(int argc, char **argv)
{
 try {
  HTTP ic;
  ic.open("127.0.0.1");
  for(int i = 1; i < argc; i++)
    ic.get(argv[i]);
 } catch(string msg) {
  cerr << "error: " << msg << endl;
 }
 return 0;
}

C:\ucw\chap7>c++ -c wininet.cpp
C:\ucw\chap7>c++ test-inet.cpp wininet.o -lwininet
C:\ucw\chap7>a index.htm
```

Further exercises:

To explore this case study further, try these extensions:

1. Implement the function `strip_filename()`. It should take a pathname like "c:\fred\alice.htm" or "/pub/dan/alice.htm" and extract "alice.htm". It must work with both DOS-style (`\`) and UNIX-style (`/`) paths.

2. Hyperlinks within a Hypertext Markup Language (HTML) file are constructed like this: `<a href="page.htm#tag">link</a>`. `#tag` is optional and should be ignored here (it is a reference to an anchor within the file). Examine an HTML file, extract all the hyperlinks, and eliminate duplicates. You can further restrict the file by extracting only links to one server. When you have such a list, you can start downloading the links. It’s important with this kind of “spider” program to know when to stop; for example, you can stop at either a fixed number of pages, a fixed number of megabytes, or a fixed depth of traversal. (This kind of program can make you very unpopular with network administrators.)

3. WinInet supplies a set of functions that enable File Transfer Protocol (FTP) access. Write a class that wraps up these functions. (The WinInet API is documented on the Microsoft MSDN Web site.)

What’s Next?

In this chapter you have learned that functions can be put inside structures, and instead of `fn(obj,args)`, you use `obj.fn(args)`. A class becomes responsible for its data, and it is given privileged access to that data. Dealing with too much detail is the chief downfall of big systems. Encapsulation keeps implementation details away from the rest of the program and means that a class can safely use different implementations. As long as the interface doesn’t change, then the client code is not broken.

In Chapter 8 you will see how C++ classes can be extended via inheritance. With inheritance, reusing code written for similar projects becomes very easy because classes can be specialized to do new things.

8

Inheritance and Virtual Methods

In the Chapter 7, “Classes,” you learned that encapsulating class data and methods together means that the rest of a program can concentrate on using that class. You learned that by using classes, you can implement abstract data types (ADTs), which present an interface to the classes’ clients. You can combine different classes together to build up new classes.

This chapter discusses *inheritance*, which is another powerful way to build families of related classes (that is, *class hierarchies*). You use these class hierarchies to model a problem. C++ objects can carry *runtime type information* (RTTI), which makes true object-oriented programming possible.

In this chapter you will learn

- How to create new classes by using inheritance
- How to initialize member objects and references
- How to use virtual methods
- How to do dynamic typecasting

The Idea of Inheritance

Different things often have something in common. Buses and cars are all four-wheeled vehicles, a goat and a sheep are both plant-eating domesticated animals, and so on. In object-oriented programming things are modeled using classes; *inheritance* is a powerful way to express the common properties of classes.

Inheritance, as you will see, will also save you a lot of typing. A class can inherit all the properties of another class, plus some extra functionality. So it is useful for customizing existing classes.

Extending a `struct`

Say you want to define an `Employee` class. In Chapter 7 you created quite a lot of code that uses the `Person` class, and so it seems reasonable just to copy the fields of `Person` and add some extra fields to create the `Employee` class: the date of employment and the employee ID number. The following is an `Employee` class that is mostly a copy of `Person`:

```
struct Employee {
    string m_name;
    long   m_phone;
    string m_address;
    long   m_id;
    //
    Date   m_employed;
    Long   m_num;

    string name()  { return m_name; }
    long   phone() { return m_phone; }
    Date   employed() { return m_employed; }
};
```

NOTE

Remember that there is no essential difference between `struct` and `class`.

Making up new classes like this can get ugly: Imagine if `Person` were a fully-fledged class with dozens of methods that would all have to be copied. Generally, copy-and-paste programming is not a good idea; it is wasteful and confusing. A common nightmare for maintainers is having sections of code that are **almost** identical, which happens when code is copied and then modified. Anytime you had new ideas about a `Person` class, you would again have to copy code. For example, I've obviously left out an email address in both my definition of `Person` and `Employee`; both classes would have to be changed.

Also, if there were a whole library of classes and functions that work with Person objects, would you also want to copy them? For instance, say there is a function print(Person *) that prints out a Person class, given a pointer. The following hack actually works:

```
Person walker;
print ( &walker );
Employee janitor;
print((Person *) &janitor);
```

You know that Person and Employee are practically identical, up to the fourth field; therefore, it is possible to pretend that an Employee class is a Person class. However, C++ does not automatically convert between unrelated classes, so a typecast (Person *) is required.

I've called this a hack because it's quick, it works, and it's a recipe for future disaster. You can get away with using the code shown in the preceding example if the structures are fairly simple. But again, any small change to the first fields causes problems. (For instance, some other programmer adds an email address to Person.) Also, you will see later in this chapter how something as innocent as adding a method can change the layout of a class in memory. As a general rule, you should not make these kinds of detailed assumptions about how the fields of classes are laid out because they might work on one machine and fail utterly on another.

Another approach is to actually put a Person object inside the Employee class, as shown in the following code. You then have access to a bona fide Person object (such as print(&janitor.m_person)) whenever you need it. The name() and phone() methods of Employee are defined using the same methods of Person. (If methods have the same name, they can be distinguished by their full names: Employee::name(), Person::name(), and so on.)

```
struct Employee {
    Person m_person;
    Date   m_employed;
    long   m_num;

    string name() { return m_person.name(); }
    long phone()  { return m_person.phone(); }
    Date employed() { return m_employed; }
};
```

This is a good solution that is commonly used. This structure does expose its innards in an unseemly way, however, in expressions such as janitor.m_ person. The need to properly encapsulate the classes' assumptions leads to many Employee methods merely calling Person methods. This extra typing is inevitably error prone; imagine if Person had dozens of methods like name() and phone().

The official C++ way of solving the problem is to *derive* the class `Employee` from the class `Person`; `Person` is said to be the *base class* of `Employee`. The base class follows the class name, separated by a colon (:)

```
struct Employee: Person {
    Date   m_employed;
    long   m_num;
};
```

After you have defined an `Employee` object, you can access fields such as `m_name` and `m_id` as if they were members of `Employee`. We say that `Employee` *inherits* `m_name` from `Person`, as well as `name()`. The ability to use the code of the *parent class* is called *implementation inheritance*. Figure 8.1 shows how the fields of `Employee` are laid out in memory: `Employee` automatically contains all fields from `Person`, plus two extra fields. The following example uses the UnderC `#d` command to show all fields of the object `cleaner`, showing that an `Employee` object contains everything from `Person`:

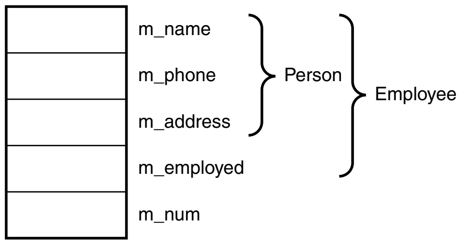


Figure 8.1: *The memory layout of the* `Employee` *structure.*

EXAMPLE

```
;> Employee cleaner;
;> cleaner.m_name = "Fred Bloggs";
;> cleaner.name();
(string) 'Fred Bloggs'
;> #d cleaner
(Date) m_employed = Date {}
(long int) m_num = 0
(string) m_address = ''
(long int) m_id = 0
(string) m_name = 'Fred Bloggs'
(long int) m_phone = 0
```

`Employee` as a Subset of `Person`

As with member functions, there is something of a stage magician's trick about inheritance; inheritance is very clever, and everyone applauds, but why go to so much trouble? In object-oriented programming, you try to model a problem so that the system reflects the world, by using objects, actions, and relationships. The classes in a program represent people, things, and abstractions. There is a very clear relationship between Employee and Person; every Employee object is a kind of Person object. In the following example, two functions have been declared, taking a pointer to Person and a reference to Person, respectively. C++ will happily allow you to pass an Employee object to these functions:

```
;> void print(Person *p);
 ;> bool read(Person& p);
 ;> Employee fred;
 ;> read(fred);
(bool) true
 ;> print(&fred);
```

Working the other way, from Person to Employee, does not happen automatically. It is true that every Employee object is a Person object, but not true that every Person object is an Employee object. In the following example, C++ requires a typecast (Employee *) to pass p to cust_id(). The answer returned by cust_id() is nonsense, for the simple reason that Person doesn't have an m_num field.

```
;> int cust_id(Employee *e)
;> { return e->m_num; }
;> cust_id(&cleaner);
(int) 0
;> Person p;
;> cust_id(&p);
CON 16:Could not match void cust_id(Person*);
0 parm
;> cust_id((Employee *)&p);
(int) 1701602080
```

You can build a chain of classes by using inheritance. For example, you can derive Manager from Employee. A Manager object has all the properties of an Employee object, plus a list of everyone managed and a salutation (Dr., Mr., Ms., and so forth).

```
struct Manager: Employee {
  list<Employee *> m_managed;
  string           m_salutation;
 };
```

Although by default Manager inherits name() from Person (as well as m_employed from Employee, and so on), it is possible for Manager to redefine name(). Assume that the company we're talking about is the old-fashioned kind where the directors are Mr. Smith, Dr. Lorenz, and so on. Their names must always be preceded by the salutation. In this code you must precisely specify Person::name(). If you left out the explicit Person scope, then the compiler would read `name()` as `Manager::name()`, which would result in an uncontrolled recursive call. Here is a definition of the `Manager::name()` method:

```
string Manager::name()
{
  return m_salutation + " " + Person::name();
}
```

In this case you can say `m_name` instead of `Person::name()`, but it is a good idea to let the base class manage its own data. If `m_name` were a private member of `Person`, you would not be able to use it directly anyway.

Inheritance makes it possible to use strong type checking for function arguments and yet flexibly pass such functions many different kinds of objects. Originally, when the code to print out the `Person` class was written, nobody was thinking of `Employee` objects. But because any `Employee` object in effect contains a `Person` object, you can reuse that printing code without forcing the type. Remember you had to use a typecast to pass a Person pointer as an Employee pointer: `cust_id((Employee *) &p)`. (This is sometimes called *type coercion,* and it is similar to a child trying to get a square peg into a round hole. It might work, depending on the particular child and the particular hole.)

Inheritance creates a lineage of related classes, and the distance along that line is the number of generations between two classes. For example, `Employee` is closer to `Person` than `Manager` is to `Person`. This distance is used when resolving overloaded functions. Here there are two versions of `print()`; `Manager *` will match the second version because the argument type is closer.

```
void print(Person *p);   //(1)
void print(Employee *e); //(2)
Employee ee; Manager mm;
print(&mm);    // will match (2), not (1)
```

Either version of `print()` works on `Manager *`, but `Employee *` is closer to `Manager *` than `Person *` is. The second version of `print()` is clearly more specialized (and it may well be defined in terms of the first version).

Access Control and Derivation

Up to now I've used `struct` instead of `class`, because it is slightly simpler. The difference when using `class` is that all the data fields are private by default, and the only access to these is through get and set methods like `employed()`:

EXAMPLE

```
class Employee: public Person {
    Date   m_employed;
    long   m_num;
  public:
    void employed(const Date& d) { m_employed = d; }
    void number(long n)          { m_num = n;      }
    Date employed()   { return m_employed; }
    long number()     { return m_num;      }
 };

typedef std::list<Employee *> EmployeeList;

class Manager: public Employee {
  EmployeeList m_managed;
  string       m_salutation;
 public:
  EmployeeList& managed()         { return m_managed; }
  void salutation(string s)       { m_salutation = s; }
  string name()
  { return m_salutation + " " + Person::name(); }
 };
```

Anything private in `Employee` will not be available to `Manager`; methods of `Manager` cannot accesss `m_num` and must use `number()` instead. This makes it much easier to change code in `Employee` without having to worry about code in all of `Employee`'s derived (or *descendant*) classes.

Making all the data private is often too restrictive; you don't want everyone outside to rely on the implementation details, but you can let derived classes have direct access with the `protected` access specifier. Here is a (nonsense) example:

EXAMPLE

```
class One {
private:
   int m_a;
protected:
   int m_b;
public:
   int m_c;
};

class Two: public One {
protected:
   int m_c;
public:
  int use_a() { return m_a; } // error: m_a is private to One
  int use_b() { return m_b; } // ok: m_b is protected
  int use_c() { return m_c; } // ok: _everyone_ can access m_c!
};
```

```
int use_One(One& o) {
  return o.m_b;        // error: m_b is protected
}
```

This is the C++ equivalent of the saying “blood is thicker than water”; it shows that relationship is privileged. Saying that `One::m_b` is protected means that any children of `One` have direct access to `m_b`. By saying that `One::m_a` is private, the writer of the class is saying that the implementation can change and that no derived class should rely on it. Note the `public` attribute in the declaration of `Two`; it shows that `One` is a *public base class* of `Two`. This is an extremely common situation. `Two` has free access to all public and protected members of `One`. (Sometimes you will see a *private base class*. This might seem like the programming equivalent of a black hole; but you can use classes with private base classes when you need full control of what appears as public.) From now on, this chapter explicitly indicates the access mode by using `class` instead of `struct` and by using public derivation.

Constructor Initialization Lists

Until this point, we have assumed that base classes have default constructors. Although there is no explicit constructor for `Employee`, the `Date` field might need to be constructed, and its base class `Person` certainly has a default constructor to make sure that the strings are properly initialized. In the section “Constructors and Destructors” in the last chapter you saw that such a constructor would be automatically supplied for `Person`. In this way, you can continue to pretend that complex objects such as strings really are simple C++ types.

In the same way, the compiler generates a default constructor for `Employee` using these constructors. It is helpful to think of the `Person` part of `Employee` as being a member object. If `Employee` had an explicit constructor, the compiler would quietly add code to that constructor to construct the base class, as well as any member objects. Therefore, the following would work well:

```
class Employee: public Person {
 ...
 public:
    Employee(string name, int id) {
      m_name = name;
      m_employ_num = id;
    }
 ...
 };
```

If the base class does not have a default constructor, then it must be called explicitly. Consider the case when `Person::Person()` takes a string argument intended to initialize m_name; the Employee constructor must call Person::Person(string name) explicitly in a *constructor initialization list,* which might also contain member initializations. This list follows a colon (:) after the constructor declaration and will contain any base class or member objects which need initialization. It is the only way you can pass arguments to the constructors of these objects. For example,

```
// have to do this with base classes
Employee(string name, int id) : Person(name)
{
  m_employ_num = id;
}
 // or ….
Employee(string name, int id)
: Person(name), m_employ_num(id) { }
```

There are two other cases in which you need to use initialization lists. First, you use initialization lists if any member objects or base classes have no default constructors. If Employee does have a Person member, then Person needs to be constructed with an explicit name value. In this case, you can think of the base class as a member.

Second, you have to use initialization lists if any members are references. In both of these cases, you cannot have a declaration without an explicit initialization. For example:

```
struct Fred {
 Person m_person;
 int& m_accessed;

 Fred(string name, int& i)
   : m_person(name), m_accessed(i)
  { }
  ...
};
```

Constants in Classes

An easy mistake to make is to try to initialize a member variable directly. This is, after all, how you usually initialize variables, and in fact Java allows this for class fields. The only place that standard C++ allows this kind of initialization is with constants, which must be declared static as well as const. But not every compiler manages this case properly, and you often see enumerations used for this purpose. In this example, you can see that generally initializations inside a struct are not allowed, except for constants. I have also shown an enum used to create constants. Both work fine (except on some older compilers like Microsoft C++ 6.0), and in both cases you access the constant with scope notation: `F1::a`, and so on. This is how constants like `string::npos` and `ios::binary` are implemented.

```
;> struct Test {
;1} int a = 1;
CON 3:initialization not allowed here

;> struct F1 {
;1} static const int a = 648;   // Microsoft C++ 6.0 complains
;1} enum { b = 2, c = 3 };      // always acceptable
;1} };
;> F1::a;
(const int) a = 648
;> F1::b;
(const int) b = 2
```

Using Classes as Exceptions

In Chapter 4, in the section "Defensive Programming," when I discussed exception handling, I mentioned that throwing different types of exceptions in functions gives any caller of those functions great flexibility in handling those different types. A function may choose to ignore some types of exceptions and let them be caught elsewhere. For instance, in the reverse-Polish calculator case study in Chapter 4, "Programs and Libraries," `string` exceptions are true errors and integer exceptions are used to signal that there is no more input; in the case of `string` exceptions, the message is printed out, but the loop is not terminated.

You can think of the `catch` block following a `try` block as a function that receives the thrown exception as its (optional) argument. A number of `catch` blocks are possible because the runtime system looks at these argument types in turn and decides which is the best match; this is a simple form of overloading. Classes make excellent types for exceptions because you can generate as many distinct, well-named types as you like. For example, the following is an example that defines an `Exception` type; it defines the single method `what()`, which returns some explanation of what error occurred (for example, `ParmError` is a kind of `Exception` error, and `DivideByZero` is a kind of `ParmError` error):

EXAMPLE

```
class Exception {
 string m_what;
public:
 Exception(string msg) : m_what(msg) {}
 string what() { return m_what; }
};
```

```
class ParmError: public Exception {
public:
  ParmError(string msg) : Exception(msg) {}
};

class DivideByZero: public ParmError {
public:
  DivideByZero() : ParmError("divide by zero") {}
};

int divide(int i, int j)
{
  if (j == 0) throw DivideByZero();
  return i/j;
}

void call_div()
{
 int k;
 try {
   k = divide(10,0);
 }
 catch(DivideByZero&) {
   k = 0;
 }
 catch(ParmError& e) {
   cout << e.what() << endl;
 }
 catch(Exception&) {
   throw;  // re-raise the exception...
 }
}
```

This set of classes classifies run-time errors. All errors will be caught by `Exception`; all errors to do with bad parameters will be caught by `ParmError`; and only dividing by zero will cause `DivideByZero`. So the calling function can choose exactly how detailed its error handling should be. It could just catch `Exception` and display its `what()` value.

The important thing to realize about using *exception hierarchies* is that you need to have the `catch` blocks in the right order. It would be wrong to put `catch(ParmError&)` before `catch(DivideByZero&)` because `DivideByZero` is a kind of `ParmError` error and its catch block would catch the exception first. This is very different from how normal function overloading works, of course.

Polymorphism

A group of related objects can have common behaviour, which still can be very distinctive. For instance, pens, pencils, and markers all can be used to draw, yet they leave very different marks on paper. Most animals make a noise, but the precise sound is unique to the animal.

Things with common behaviour can be modelled by using inheritance; all classes that share a base class will inherit that base classes' behaviour. A group of classes related by inheritance is called a *class hierarchy*.

If you have a number of graphical objects, they all will have at least one thing in common: They can draw themselves. That is, they will all have a `draw()` method. If these graphical objects all derive from some common base class `Shape`, which defines `draw()`, then they can all be accessed through `Shape::draw()`.

Polymorphism, from the Greek words for *many* and *forms*, means that a common function name (such as `draw()`) can have different meanings for different objects. You have already seen polymorphism in action; operator overloading makes `+` mean very different operations (like integer or floating-point addition, or string concatenation) which nevertheless are all kinds of addition. This is sometimes called *static* polymorphism, as opposed to *dynamic* polymorphism, when the actual method to call is decided only at run-time. How this magic works and how it can work for you is discussed in the following sections.

Class Hierarchies

Consider the family tree shown in Figure 8.2. This tree is unlike human family trees in two ways: A person has two parents, not one, and generally a family tree is concerned with the patrilineal line only. Similarly to the way humans inherit from their ancestors, class inheritance creates a *class hierarchy*. As shown in Figure 8.2, both `Temp` and `Manager` are derived from `Employee`. However, Figure 8.2 does not show a hierarchy of importance; it doesn't mean than `Temp` and `Manager` are equally important, and less important than `Employee`. Rather, a class hierarchy is like a classification of animals, which is based on the animals' evolutionary ancestors.

When a program is running, objects form dynamic relationships with each other (for example, `Manager` keeps a list of `Employee` objects) called the *object hierarchy*, in which a `Manager` object is indeed more important than a `Temp` object.

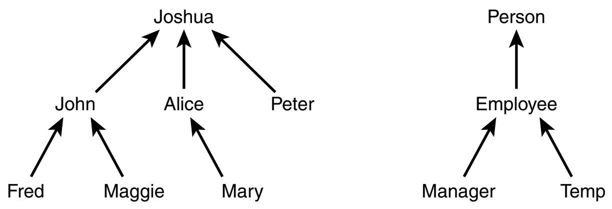


Figure 8.2: *A family tree and a class hierarchy.*

Class hierarchies depend on what you are trying to model. Figure 8.3 shows a fairly arbitrary classification of some domestic animals; the basic division is according to diet (that is, carnivore, herbivore, omnivore). This hierarchy might be more useful to a farmer or zookeeper than a rigorous genetic classification, which would put elephants and shrews next to each other.

The Animals hierarchy also classifies a `Person` as an omnivorous animal. This is a completely different view of `Person`, which would not be useful for human resources or payroll applications.

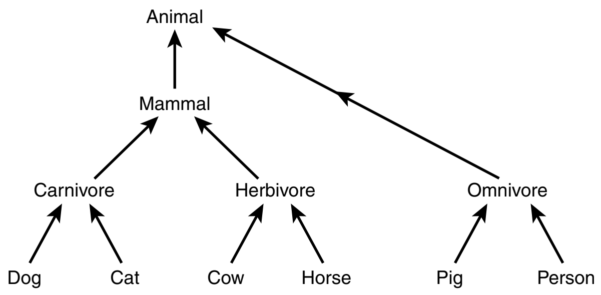


Figure 8.3: *A classification of animals by diet.*

NOTE

A C++ class can have more than one base class. Strictly speaking, a cat is both a mammal and a carnivore, and many nonmammals are carnivorous. It is equally true that an `Employee` object is a `Person` object and is also a `TaxPayer` object. It is derived from two parent classes, just as people are derived from two parents. However, not everyone thinks that *multiple inheritance* is good object-oriented design. It is certainly important to get single inheritence right first. So this book does not discuss multiple inheritance.

A Hierarchy of Animals

Consider the following example, which defines the classes `NamedObject`, `Animal`, and `Dog`:

```
class NamedObject {
private:
  string m_name;
public:
  NamedObject(string name)
  : m_name(name) {}
  string name()  { return m_name; }

};

class Animal: public NamedObject {
public:
  Animal(string name)
  : NamedObject(name) {}

  static void say(std::string s) {
    cout << "say: " << s << endl;
  }

  void call_out() { say("<nada>"); }
};

class Dog: public Animal {
private:
  string m_breed;
public:
  Dog(string breed = "mongrel")
  : Animal("dog"), m_breed(breed) {}

  string breed()  { return m_breed; }

  void call_out() { say("woof!"); }
};

void exercise_animal(Animal* pa)
{
  pa->call_out();
  if (pa->name() == "dog") {
    cout << "breed: " << ((Dog *)pa)->breed() << endl;
  }
}
;> Animal a("?"); Dog d;
;> a.call_out();
<nada>
;> d.call_out()
woof!
```

```
;> exercise_animal(&d);
<nada>
breed: mongrel
```

An `Animal` object has a name; you can move this property of `Animal` into the base class `NamedObject`, which will not only simplify `Animal` but any other classes that carry names. Doing this also means you could in future impose some policy on all names in your system; for instance, that names should not contain any special characters. Note that `NamedObject` has a single constructor taking a `string` argument, so `Animal` must call this constructor using an initialization list.

`Animal` has a method `call_out()` that you use to print out the `Animal` object's cry. `Animal::call_out()` does nothing specific, because this is a general class. `Animal` also supplies a function `say()` for speaking to the world, which is `static` in this case because it doesn't depend on the particular object.

The `Dog` class is derived from the `Animal` class, which means it inherits everything that is publicly defined in `Animal`; that is, it inherits `name()`, `say()`, and `call_out()`. You redefine `call_out()` because a `Dog` object is a definite kind of `Animal` object that makes a definite sound. The `Dog` class can also have a breed value, although it defaults to being a mongrel.

Both generic `Animal` objects and specific `Dog` objects have a `call_out()` method, but they are really different functions that have the same name. Usually C++ makes a final decision at compile time about what function to call; this is called *early binding*. The `exercise_aninal()` function only knows about the plain generic `Animal::call_out()`, and so `Dog::call_out()` is not called.

Notice the expression `((Dog *)pa)->breed()`; the parentheses are necessary because `operator->` has a higher precedence than the typecast operator `(Dog *)`. Besides being clumsy, it is a dangerous operation to apply to `Animal` objects that are not `Dog` objects, because they don't have an `m_breed` field. In this case, coercing or forcing the type is likely to cause grief. (This is why you had to explicitly test for the `Animal` object's name.) Such typecasts are called *static* because they happen at compile time and are basically dumb. Because these typecasts are potentially unsafe, modern C++ uses the `static_cast` keyword. The following example shows `exercise_animal()` rewritten to properly call the appropriate `call_out()` method; the principle here is that you make dangerous things obvious—it's much easier to search code for `static_cast` than for C-style typecasts like `(Dog *)`:

```
void exercise_animal(Animal *pa)
{
  if (pa->name=="dog") {
    Dog *pd = static_cast<Dog *>(pa);
    pd->call_out();                   // Dog::call_out
    cout << "breed: " << pd->breed() << endl;
  }
  else pa->call_out();               // Animal::call_out
}
;> exercise_animal(&d);
woof!
breed: mongrel
```

This is not a very satisfying function. Sooner or later, somebody will need to keep track of `Horse` objects, and another `if` statement will have to go into `exercise_animal()`. The problem is worse than it seems in this simple example; there are likely to be many such functions, and they all have to be modified if extra animal classes are added. Unless you get paid for each line of code you write (and enjoy debugging bad code all night), you should not choose to go down this route. Yes, we could make the animal's sound a member field, but that would not solve the general problem of making animals with different behaviors.

It would be better if calling `Animal::call_out()` would automatically select the correct operation. How this is done and how it works is the subject of the next section.

Virtual Methods

Let's look at the classes `Animal` and `Dog` again, with a one-word change: adding the qualifier `virtual` to the first definition of `call_out()`. You then create an `Animal` class and a `Dog` object, and you can make two `Animal` pointers that refer to them:

EXAMPLE

```
class Animal: public NamedObject {
public:
  Animal(std::string name)
  : NamedObject(name) {}

// notice that call_out() has become virtual
  virtual void call_out()
  { say("<nada>"); }
};

class Dog: public Animal {
private:
  string m_breed;
public:
  Dog(string breed)
  : Animal("dog"), m_breed(breed) {}
  string breed()  { return m_breed; }

  void call_out()  // override Animal::call_out
 { say("woof!"); }
};

void exercise_animal(Animal *pa)
{
  cout << "name: " << pa->name() << endl;
  pa->call_out();
}
;> Animal a = "?"; Dog d;
;> Animal *p1 = &a, *p2 = &d;
;> p1->call_out();
<nada>
;> p2->call_out();
woof!
;> exercise_animal(&a);
name: ?
<nada>
;> exercise_animal(&d);
name: dog
woof!
```

This example defines yet another version of `exercise_animal()`, which only calls that which is common to all animals, that is, their ability to call out. This time, calling `call_out()` calls the correct function!

The method `call_out()` is called a virtual method, and redeclaring it in any derived class is called *overriding* the method. (You can use the `virtual` qualifier on the overridden method, but it isn't necessary.) It is a good idea, however, to use a comment to indicate the fact that a method is overridden. It is important that the overriden method be declared with the same signature as the original; otherwise, you get an error (if you're lucky) or a warning about the fact that the new function is hiding a virtual function. That is, if the original declaration was `void call_out()`, then you should not declare it as `int call_out()` in some derived class, and so forth.

NOTE

If you have previously used Java, note that Java makes all methods virtual by default, unless you explicitly use the keyword `final`.

Here is another class derived from `Animal`:

```
class Horse: public Animal {
public:
  Horse()
  : Animal("horse") {}
  void call_out() // override Animal::call_out
  {
   say("neigh!");
  }
};
;> Horse h;
;> exercise_animal(&h);
name: horse
neigh!
```

You can add new animal objects, and any function that understands animal objects will call the correct method. Similarly, it can be useful to make `say()` virtual. What if the animals were expected to announce themselves in a window rather than to the console? In this case, you simply override `say()`, and all subsequent classes output to windows.

How does the system know at runtime what method to call? The secret to *late binding* (as opposed to early binding) is that any such object must have *runtime type information* (RTTI). Virtual methods are not identified by an actual address, but by an integer index. The object contains a pointer to a table of function pointers, and when the time comes to execute the method, you look up its actual address in the table, by using the index. (This is shown in Figure 8.4.) The table is called the *virtual method table* (VMT), or *vtable*, and every class derived from `Animal` has a different VMT, with at least one entry for `call_out()`. Such classes are bigger than you would expect because there is always some allocated space for the hidden pointer. The C++ standard does not specify where this pointer is found; in some compilers (such as the Microsoft and Borland C++ compilers), it is the first field; in GNU C++ it is the last field; and in UnderC it is just before the first field. This is the main reason you should not depend on a particular layout of complex classes in memory; the original definition of `Employee` would not match `Person` if you added even one virtual method. Virtual methods are also slightly slower because of this lookup, but you should be aware that the main performance issue is that making everything virtual makes inlining impossible (that is, it cannot insert code directly instead of calling a function.)

Classes that contain virtual methods are called polymorphic; this refers to how `call_out()`, for example, can be redefined in many different ways.

You often need to keep collections of polymorphic objects. First note how assignment works between such objects. This example assigns a `Dog fido` to an `Animal cracker`:

```
;> cracker = fido;    // Dog => Animal is cool…
;> cracker.call_out();
```

```
<nada>
;> Animal *pa = &fido;
;> pa->call_out();
woof!
```

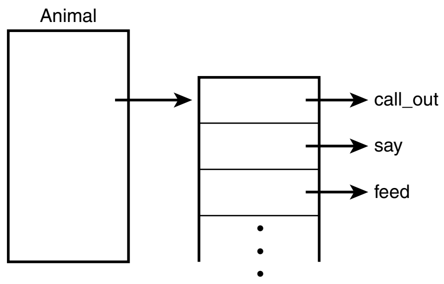


Figure 8.4: *Each `Animal` object has a hidden pointer to a VMT.*

The type conversion in this example works, but it throws away information because a `Dog` object is larger than an `Animal` object (that is, it contains extra breed information). Plus—and this is very important—`cracker` remains an `Animal` object, with a hidden pointer to an `Animal` VMT. Any assignment between `Animal` and `Dog` will have this result. If there was a list of `Animal` objects, then adding `Dog` objects would involve such an assignment. Therefore, the following is not the way to make a collection of `Animal` objects:

```
;> list<Animal> la;
;> la.push_back(cracker);
;> la.push_back(fido);
;> la.back().call_out();
<nada>
```

To make a collection of `Animal` objects, you should keep a list of pointers to `Animal` objects. As Figure 8.5 shows, if you do this, it is no longer a problem that some `Animal` objects are larger objects than others. In the next example, various objects are created with `new` and added to the list with `push_back()`. I then define a function `animal_call(Animal *pa)` and use the standard algorithm `for_each()` to call this function for each element in the list:

EXAMPLE

```
;> list<Animal *> zoo;
;> zoo.push_back(new Animal("llama"));
;> zoo.push_back(new Dog("pointer"));
;> zoo.push_back(new Horse());
;> void animal_call(Animal *pa) { pa->call_out(); }
;> for_each(zoo.begin(),zoo.end(),animal_call);
<nada>
woof!
Neigh!
```

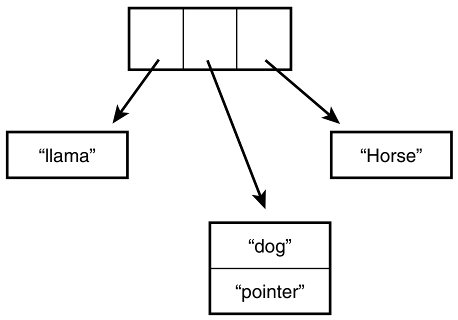


Figure 8.5: *A list of pointers to `Animal` objects.*

Any `Animal` pointer might be a pointer to a `Dog` object. A type field such as `name()` makes the identification easier, but this is irritating to set up. C++ provides an interesting typecast operator called `dynamic_cast`, which you can use for polymorphic classes. `dynamic_cast<A *>(p)` will use RTTI to decide at run-time whether `p` is in fact a `A *` pointer. If not, it will return NULL. Note that any class derived from A will also qualify. Here it is in action:

EXAMPLE

```
;> Animal *p1 = new Horse, *p2 = new Dog;
;> dynamic_cast<Dog *>(p1);
(Dog *) 0
;> dynamic_cast<Dog *>(p2);
(Dog *) 9343e1a0
;> bool is_dog(Animal *a)
{ return dynamic_cast<Dog *>(a) != NULL; }
;> struct Spaniel: public Dog { Spaniel() { breed("spaniel"); } };
;> is_dog(new Spaniel);  // A Spaniel is a kind of Dog…
(bool) true
;> // remove the dogs from the zoo!
;> // (std::remove_if() - see Appendix B, "Standard Algorithms")
;> remove_if(zoo.begin(),zoo.end(),is_dog);
```

Note that `dynamic_cast` does not work on just any type. It is specifically designed to exploit the fact that classes with virtual methods always carry extra type information. Without `dynamic_cast`, you need to put in the extra type field by hand. It is always safer than `static_cast`, which does not check the actual type at run-time.

Abstract Classes

In a way, there is no such thing as a plain `Animal` object. In the real world, any `Animal` is a specific species or at least has a well-defined family. It makes no sense to create an `Animal` class on its own, but only to create specific derived classes like `Dog`, `Horse`, and so forth.

`Animal` can be used as an *abstract base class*; the method `call_out()` is initialized to zero, which means that this virtual method is not defined yet (that is, it is a *pure virtual method*). A class with a number of pure virtual methods cannot be created directly, but only used as a base class. Any derived class must supply a proper definition of `call_out()`. As you see in this definition of `Animal`, `call_out()` is not given any definition:

```
class Animal: public NamedObject {
public:
  Animal(std::string name)
  : NamedObject(name) {}

  virtual void call_out() = 0;
};

;> Animal *pa = new Animal("sloth");
CON 32:class contains abstract methods
```

Code Reuse

By now you may have gotten the impression that object-orientated programming (OOP) often involves a lot of typing. But this is because OOP is not just about writing the current program, but the next version, and any similar programs. Only the smallest programs have no future development path, so it pays to take a little more trouble. OOP aims to provide a good foundation for further development.

Making code more reuseable is a practical way of improving programming productivity. Classes are often easier to reuse than libraries of plain functions (although they still have their place.) For example, it is easier to use `string` than to use the C-style string manipulation functions like `strcpy()`, and so forth.

Class frameworks are effectively customizeable applications. By modifying the framework's behaviour with inheritance, you can write powerful graphical user interface (GUI) programs with a single page of code. This has probably been the most successful kind of object-oriented code reuse.

Reusable Objects and Functions

A goal of software engineers is to build the best possible system with the least amount of effort. There is no intrinsic value in hard work; the definition of efficiency in engineering is the amount of work you can get done with a given expenditure of energy. Sir Isaac Newton said that if he saw further than most, it was because he stood on the shoulders of giants; we can borrow from Newton to say that many giants have labored to produce excellent software libraries. Many of the existing libraries are free, but even if you do have to pay for them, remember that reinventing the wheel is nearly always more expensive. It takes a lot of effort to create a good new C++ `string` class, and the community of programmers will not thank you for doing things differently from everyone else. The standard library will often have all you need.

Code reuse is about as old as programming itself. Reusing routines or functions works very well for well-defined mathematical operations such as the trigonometric functions. Any working programmer soon develops a collection of useful functions (for example, functions for trimming strings or filename extensions), and it would be silly to wrap such things up in a class. In such a case, it is more appropriate to use namespaces.

In object-oriented software development, the class is the unit of reuse. Rather than a loose collection of functions for manipulating character strings (as in C), C++ provides the standard `string` class. The rest of the program (that is, the *client code*) can ignore the details, trust the implementor, and deal with a convenient high-level abstraction. It is worth putting effort into producing high-quality software components because they can make more than one project easier.

Using an Existing Framework

Consider the following (silly) abstract class, which again deals with the general problem of computers washing their hair:

EXAMPLE

```
class HairNight {
public:
  virtual void find_shampoo() = 0;
  virtual void lather() = 0;
  virtual void rinse() = 0;
  virtual int no_of_times() = 0;
  virtual void dry_hair() = 0;

  void wash_hair()
  {
    find_shampoo();
    for(int i = 0, n = no_of_times(); i < n; i++) {
       lather();
       rinse();
    }
    dry_hair();
  }
};
```

Because this class is abstract, you cannot construct a real object of this type, but only use it in derivation. `HairNight` contains no information about how to find the shampoo or wash the hair, but it knows how to put all these together. A real derived class has to fulfill the contract and supply real versions of the pure virtual functions. This is the essence of a *class framework*; it relies on clients to adapt its abstract functionality.

The most common use for class frameworks is graphical user interface (GUI) applications. There is no standard C++ library for handling graphics and GUI applications. Many of the class frameworks used for GUI development, particularly on PCs, are both proprietary and odd. But the same concepts are common to all GUI application frameworks: GUI applications supply you with an application, and you customize it by overriding virtual methods.

NOTE

Hopefully, when the International Standards Organization committee reconsiders the C++ standard in a few years, there will be sensible proposals for windowing classes and grapics, in the same way that the (Abstract Windowing Toolkit) AWT is standard for Java.

YAWL

The original purpose behind Yet Another Windows Library (YAWL) was to take the tedium out of Windows programming. A small class framework is more useful in demonstrating GUI programming than large proprietary beasts such as the Microsoft Foundation Classes (MFC) and Borland's Visual Control Library (VCL). With YAWL, you can produce a standalone 60KB executable GUI program.

A basic GUI application can inherit most of its behavior from a YAWL application window. For example, consider the business of drawing trees, which is discussed in Chapter 6, in the section "Drawing Trees with Turtle Graphics." One serious downside of that style of turtle graphics is that the output is not persistent and would disappear if the window were resized or other changes were made. Here is the YAWL version of a program in which the window can be resized arbitrarily and the tree will be redrawn in scale:

EXAMPLE

```
#include <tg-yawl.h>
#include <sstream>
using namespace std;

#include "tree.h"

class MyWin: public TGFrameWindow {
public:
 void tg_paint(TG& tg)              // override
  {
// starting in the middle, draw the tree
   tg.go_to(50,10);
   draw_tree(tg,20,0.7,ang,0.1);
  }

 void tg_mouse_down(FPoint& p)     // override
 {
   ostringstream out;
   out << '(' << p.x << ',' << p.y << ')' << ends;
//  change the window caption
   set_text(out.str().c_str());
 }
};

int main()
{
  MyWin w;
  w.show();        // make the window visible
  w.run();         // respond to events
  return 0;
}
```

This short YAWL program demonstrates the basic principle of class frameworks. Our class `MyWin` derives from `TGFrameWindow`, and overrides the methods `tg_paint()` and `tg_mouse_down()` to do application-specific work. First, an application frame window is constructed and made visible through the use of `show()`; the application starts processing events when `run()` is called. When the user closes the window, `run()` returns, and the program ends.

To do anything useful, you need to respond to events (often called *messages*) from the operating system and from the users. People often have difficulty with event-driven programming because it is so foreign to the usual way of doing programming. People are accustomed to being fully in control, able to issue commands when they choose—at least in the domain of software. Busy personal assistants do not have this luxury; they must be prepared to drop anything to answer the phone, and generally their work priorities are externally determined. Real-world programs are like busy personal assistants; they must take their cues from the environment and be prepared to multitask.

The application window, for instance, may be called on to repaint itself at any point. The framework calls the `tg_paint()` method when this happens, passing it a valid turtle graphics object. Because `tg_paint()` is virtual, if your window class overrides `tg_paint()` then you can run any graphics code you like whenever the window needs to repaint itself. In a similar way, when the user clicks on the window, the `tg_mouse_down()` method is called and will be passed the scaled coordinates.

When to Use Object-Oriented Programming

You can use any user-defined type as a base class; it does not make sense to extend or specialize ordinary numbers, in the same way that you cannot overload operators for ordinary numbers. The reason is that people already have a good idea of the behavior of numbers, and it would be upsetting to change this. Similarly, you should treat `std::string` and the standard containers as if they were built-in types.

Object-orientation is like the efficient organization of a business; ideally, everyone should have clearly defined roles. When the business expands, the responsible classes can expand their roles without being distracted by other duties. It is important to plan ahead for expansion, for both businesses and software systems. Therefore, object-oriented designers spend a lot of time on design because they are already thinking of the next version. As a group of programmers builds up a body of reusable classes, any similar projects become much easier. For example, if you are building human resources systems, an `Employee` class (especially if it is readily extensible) can save you a lot of coding.

The extra investment involved in object-orientation is not always needed. If you are building a small command-line utility, for example, it is usually not necessary. Sometimes functions are the best solution. Unless you might later decide to vastly expand the utility's functionality and build a GUI application; then object-orientation will save the day.

Case Study: The Drawing Program Revisited

In Chapter 5, "Structures and Pointers," the case study was a program to do drawings of common shapes. That program used a `struct` to keep information about each graphics shape. Unfortunately, this does not work very well if the shape isn't a simple rectangle or ellipse that can be described by two points, the lower left and upper right corners. You would have to put in a pointer, which might contain extra point data. So the first problem is how to force complicated shapes into a simple `struct`.

The second problem has to do with program organization. The previous program relied on `case` statements, and this gets awkward when shapes have other properties as well. For example, users would like to select graphics shapes interactively. This would invariably lead to the need for a new `case` statement. Adding new types of shapes means that code in several places must be changed.

This case study shows how a class hierarchy of graphics shapes can produce a straightforward design that is easy to extend.

The first important part of designing any application is to think about the classes required. In this case, you need a window object and a number of shape objects. All shapes have certain properties in common; they are required to draw themselves, and they must be able to read and write themselves to disk. The class `Shape` is abstract and defines an interface that all real `Shape` objects must supply. That is, any real graphics shape will be derived from `Shape` and override methods like `draw()` and `read()` (you will find all this code in `chap8\shapes.h, chap8\shapes.cp`, and `chap8\window.cpp`):

```
class Shape {  // an abstract base class for all Shapes...
public:
  enum { MANY = 999 };
  Shape() {}
  virtual int type() = 0;
  virtual void create(FPoint pts[], int n) = 0;
  virtual void draw(TG& tg) = 0;
  virtual int distance(const FPoint& p) = 0;
  virtual int npoints() = 0;
  virtual void read(std::istream& is) = 0;
  virtual void write(std::ostream& os) = 0;
};
```

Abstract base classes such as `Shape` are useful because they concentrate attention on the interface rather than on the implementation. The `npoints()` method is intended to return the number of points necessary; if it returns `MANY`, the object has an indefinite number of points (like a polyline). The `distance()` method is used so that an application can detect the closest object to a point. Any `Shape` object should supply a unique integer `type()` function. (Just for a change, this example does not make the `std` namespace global, so all `std` entries like `std::string` must be fully qualified names.)

As it stands, `Shape` is pure promise and no delivery. Because shapes like rectangles and ellipses can be described by a rectanglar bounding box, it makes sense to specialize `Shape` for doing such shapes, as in the following code:

EXAMPLE

```
class ShapeRect: public Shape {
protected:
  FPoint m_bottom_left, m_top_right;
public:

  void create(FPoint pts[], int n)
  {
     if (n != 2) throw Error("Supply two points for objects");
```

```
    m_bottom_left = pts[0];
    m_top_right = pts[1];
  }
  int distance(const FPoint& p)
  {
    return p.x > m_bottom_left.x && p.y > m_bottom_left.y
          && p.x < m_top_right.x && p.y < m_top_right.y;
  }
  int npoints()
  { return 2; }

  void read(std::istream& is)
  {
    is >> m_bottom_left.x >> m_bottom_left.y
       >> m_top_right.x, m_top_right.y;
  }

  void write(std::ostream& os)
  {
    os << m_bottom_left.x << ' ' << m_bottom_left.y
       << m_top_right.x   << ' ' << m_top_right.y << std::endl;
  }

 };
```

In this example, the object's bounding rectangle is represented by two points, and the object knows how to read and write bounding rectangles. The `distance()` method is fairly crude; it is set to `1` if the object is inside and `0` if the object is outside the rectangle. The `create()` method simply creates the bounding rectangle from two points and throws an exception if given any other number of points. But `ShapeRect` does not override `draw()`.

Since `ShapeRect` still does not specify how the object displays itself, it does not completely fulfill the contract. It is still an abstract class. The actual classes derived from `ShapeRect`, `Rectangle`, and `Ellipse`, (as shown in the following code) share everything in common except how to draw themselves. This is an excellent example of how using a common base class can simplify classes.

Note that these actual shapes have static `New()` member functions, which are used for dynamic creation of these objects. That is, `Rectangle::New()` creates a pointer to a `Rectangle` object. This may seem unnecessary because `new Rectangle()` does the job more simply, but you will see how useful this is when used with the `type()` method. Also, these classes have explicit default constructors that apparently do nothing. This is an UnderC limitation; within methods you can refer only to other members if they have been previously defined. `new Rectangle()` implicitly calls `Rectangle()` so I have to make sure that the default is defined first.

```
class Rectangle: public ShapeRect {
 public:
   Rectangle() {}
   static Shape *New() { return new Rectangle(); }
   int type()          { return 1; }

   void draw(TG& tg)
   { tg.rectangle(m_bottom_left.x,m_bottom_left.y,
                  m_top_right.x, m_top_right.y);
   }
 };

 class Ellipse: public ShapeRect {
 public:
   Ellipse() {}
   static Shape *New() { return new Ellipse(); }
   int type()          { return 2; }

   void draw(TG& tg)
   { tg.ellipse(m_bottom_left.x,m_bottom_left.y,
                  m_top_right.x, m_top_right.y);
   }
 };
```

There is another kind of `Shape` object you can implement that does not fit into the rectangle mold. A polyline consists of an arbitrary number of points—rather than just two—joined by lines. Because the representation of polylines is so different from the representation of regular lines, you must define special `read()` and `write()` methods for them, as in the following example:

EXAMPLE

```
typedef std::list<Fpoint> PointList;
..
class PolyLine: public Shape {
   PointList m_points;
 public:
   PolyLine() {}
   static Shape *New() { return new PolyLine(); }
   int type()          { return 3; }

   void create(FPoint pts[], int sz)
   {
    if (sz < 2) throw Error("at least one point");
    for(int i = 0; i < sz; i++)
      m_points.push_back(pts[i]);
   }
   int npoints() { return MANY; }  // indefinite no. of points!

   void draw(TG& tg)
   {
      PointList::iterator pli;
      tg.penup();
      for(pli = m_points.begin(); pli != m_points.end(); ++pli)
         tg.plot(pli->x,pli->y);
    }

   int distance(const FPoint& p)
   { return MANY; }  // for now

   void read(std::istream& is)
   {
    int n;
    FPoint p;
    is >> n;
    for(int i = 0; i < n; i++) {
       is >> p.x >> p.y;
       m_points.push_back(p);
    }
   }

   void write(std::ostream& os)
   {
     PointList::iterator pli;
     os << m_points.size() << std::endl;
     for(pli = m_points.begin(); pli != m_points.end(); ++pli)
        os << pli->x << ' ' << pli->y << std::endl;
   }

};
```

This class has a completely different representation than the other `Shape` classes, but it fulfills the same contract.

A collection of `Shape` objects needs a responsible adult to look after them and make sure they are fed and clothed. The window class `ShapeWindow` could do this in simple cases; you could keep and maintain a simple list of `Shape` pointers. However, it is better if the `ShapeWindow` concentrates on doing window-like things, such as managing the user interface, and delegates the management of `Shape` objects to `ShapeContainer`.

The `ShapeContainer` class need not derive from any class, and it would mostly contain the actual list of shapes. However, the example shows

`ShapeContainer` derived from `Shape` for a number of reasons. It can draw itself (by drawing all the shapes); it can read and write itself (by asking all the shapes to read and write themselves); and `distance()` can be interpreted as the minimum distance to a `Shape` object. Plus, it is very common for complex drawings to be built of composite shapes; a child's drawing of a house is a triangle on top of a rectangle, plus a few rectangles for doors and windows. So real drawings could be composed of a number of `ShapeContainer` shapes. This case study won't add this functionality, but it would not be difficult to implement.

The following is the class definition for `ShapeContainer`; apart from the usual `Shape` methods, you can add shapes to the container and register new kinds of shapes by using `add_shape()`:

```
// shape-container.h
#include "shapes.h"
typedef std::list<Shape *> ShapeList;
typedef Shape * (*SHAPE_GENERATOR)();

class ShapeContainer: public Shape {
private:
  ShapeList m_list;
  ShapeList::iterator p, m_begin, m_end;
  SHAPE_GENERATOR m_shape_generator[40];
public:

 void add_shape(int id, SHAPE_GENERATOR shg);
 void add(Shape *obj);

 // Shape Interface!
 void draw(TG& tg);
 void read(std::istream& is);
 void write(std::istream& os);
 int distance(const FPoint& p);

// no useful purpose so far, but required of a Shape.
void create(FPoint pts[], int n) { }
int npoints() { return MANY; }
int type() { return 0; }
};
```

Adding and drawing shapes is straightforward; in this example, the iteration over all `Shape` objects is simplified by keeping the start and finish iterators for the list of shapes. Provided that everyone uses the `add()` method and doesn't manipulate `m_list` directly, this works fine. Both `draw()` and `write()` call their corresponding method for all objects; in addition, `write()`

writes out the type value for each object. Here are the definitions of ShapeContainer's methods:

```
void ShapeContainer::add(Shape *obj)
{
   m_list.push_back(obj);
   m_begin = m_list.begin();
   m_end = m_list.end();
}

void ShapeContainer::draw(TG& tg)
{
    for (p = m_begin; p != m_end; ++p)
        (*p)->draw(tg);
}
void ShapeContainer::write(std::istream& os)
 {
    os << m_list.size() << endl;
    for (p = m_begin; p != m_end; ++p) {
        os << (*p)->type() << ' ';
        (*p)->write(os);
    }
 }
```

Before we can talk about the `read()` method, we should talk about why you need to write out the type value and why you then need those static `New()` functions. (Remember that a *static* member function is just a plain function that is defined inside a class scope, just like inside a namespace.) Here is code that shows a file consisting of a Rectangle and an Ellipse being read:

```
;> ifstream in("first.shp");
;> int i;
;> Shape *p;
;> in >> i;    // should be 1, which is type of Rectangle
;> p = new Rectangle();   // or you can say Rectangle::New()
;> p->read(in);
;> in >> i;   // will be 2, which is type of Ellipse
;> p = new Ellipse();
;> p->read(in);
```

This technique works beautifully, because the appropriate code for reading in the object will be called; `read()` is a virtual method. But you do need to construct the particular object first. How is this done?

The `read()` method of ShapeContainer first reads the type ID (as shown in the following code), and then it looks up the function needed to create the actual `Shape` object. `ShapeContainer` keeps a simple array of function pointers;

these functions take no argument, and they return a pointer to a `Shape` object. These functions are sometimes called *object factories*.

EXAMPLE

```
typedef Shape * (*SHAPE_GENERATOR)();

void ShapeContainer::add_shape(int id, SHAPE_GENERATOR shg)
{
 m_shape_generator[id] = shg;
}
…
void setup_shapes(ShapeContainer& sc)
{
  sc.add_shape(1,Rectangle::New);
  sc.add_shape(2,Ellipse::New);
  sc.add_shape(3,PolyLine::New);
}

…
void ShapeContainer::read(std::istream& is)
{
   int n,id;
   Shape *obj;
   is >> n;
   for(int i = 0; i < n; i++) {
     is >> id;
     obj = (m_shape_generator[id]) (); // call object factory
     obj->read(is);
     add(obj);
   }
 }
```

The problem here is that you need to create the specific `Shape` object. After you do that, you can call the specific `read()` method to fully construct the object. If you call the correct object generator function using the unique ID, then `read()` will call the specific `read()` method.

The `setup_shapes()` function adds the various shape generators to the `ShapeContainer` object. Notice that `setup_shapes()` is not a member of `ShapeContainer` because `ShapeContainer` need not (and should not) know what all the possible shape objects are; otherwise, you could have just have used a `switch` statement within `read()`.

The last `ShapeContainer` method computes the minimum distance of all the objects. Although we haven't yet dealt with selection, the last `ShapeContainer` method shows how the currently selected object can be found. Given a point, the distance to each graphics shape is calculated, and the closest shape is selected:

```
int ShapeContainer::distance(const FPoint& pp)
 {
    iterator p_selected = m_end;
    int d, minv = MANY;
    for (p = m_begin; p != m_end; ++p) {
      d = (*p)->distance(pp);
      if (minv < d) {
        minv = d;
        p_selected = p;
      }
    }
    if (p_selected != m_end) {
      //...do something with the selected object!
    }
    return minv;
 }
```

The window class `ShapeWindow` manages the user interface. Up to this point, you have not specifically needed the YAWL framework; you could hook turtle graphics into any framework of choice. The following example uses `TGFrameWindow` as a base class because it provides a standalone application frame window that supports turtle graphics:

EXAMPLE

```
const int NP = 60;

class ShapeWindow: public TGFrameWindow {
private:
   ShapeContainer m_shapes;
   Shape *m_obj;
   int m_count;
public:
   ShapeWindow()
   { m_count = 0; m_obj = NULL; }

   void add_point(FPoint& pt, bool last_point)
   {
     static FPoint points[NP];
     if (m_obj == NULL) {
        message("Select an object type");
        return;
     }
     points[m_count++] = pt;

     if (m_count == m_obj->npoints() || last_point) {
      try {
          m_obj->create(points,m_count);
      } catch(Error& e) {
          message(e.what().c_str());
          return;
       }
       m_shapes.add(m_obj);
       invalidate(NULL);
       m_obj = NULL;
       m_count = 0;
     }
   }

   void do_read()
   {
     TOpenFile dlg(this ,"Open Shape File");
     if (dlg.go()) {
        ifstream in(dlg.file_name());
        m_shapes.read(in);
     }
   }

   void do_write()
   {
     TSaveFile dlg(this ,"Save Shape File");
     if (dlg.go()) {
        ofstream out(dlg.file_name());
        m_shapes.write(out);
     }
   }


   void keydown(int vkey)                // override
   {
     switch(vkey) {
     case 'R': m_obj = new Rectangle(); break;
     case 'E': m_obj = new Ellipse();  break;
     case 'P': m_obj = new PolyLine(); break;
     case 'O': do_read();             break;
     case 'W': do_write();            break;
     default: message("not recognized");
     }
   }

   void tg_mouse_down(FPoint& pt)        // override
   {  add_point(pt,false); }

   void tg_right_mouse_down(FPoint& pt)  // override
   { add_point(pt,true);  }
   void tg_paint(TG& tg)                 // override
   {  m_shapes.draw(tg);  }
};
```

The ShapeWindow class inherits all the behavior of an application frame window, but it overrides the painting, keyboard, and mouse methods. The user interface is very crude: For example, the user presses the R key for a new rectangle. A Shape object of the correct type is constructed and saved in the variable m_obj, although it isn't yet fully created.

You create the object by entering the points with the mouse; when the required number of points have been entered, you can call the create() method for the object, add the object to the shapes list, and refresh the window by calling invalidate(). In the case of polylines, an indefinite number of points can be entered, and you can right-click to add the last point. Both left and right mouse events are passed to add_point(), which keeps count of the number of points that have been entered and saves them into a static array of points. Because it's static, this array will keep its values after each call to add_point().

Finally, reading and writing is a simple matter of calling the appropriate file dialogs.

All the user interface information in this example is in ShapeWindow, and all the shape management is in ShapeContainer. These roles are kept distinct because as you add features to the program, you want to keep things as simple as possible. ShapeContainer was not necessary, but it made the program structure easier to understand.

Object-oriented designs are easy to extend. For example, consider what is involved in adding a new Shape object. It is possible to create a circle by choosing the bounding points carefully, but really you need another way of doing this. You should instead construct a circle by specifying the center and a point on the radius, which for simplicity we'll assume is along a horizontal line. The Circle shape object is obviously just an Ellipse shape object, but it has a different create() method, which rearranges the points and passes them to Ellipse::create(); this is a case where the fully qualified name is necessary. Otherwise this create() will call itself indefinitely!

```
class Circle: public Ellipse {
 public:
  Circle() {}
  static Shape *New() { return new Circle(); }
  int type()          { return 4; }

  void create(FPoint pts[], int sz)
  {
    FPoint p1 = pts[0], p2 = pts[1];
    double radius = p1.x - p2.x;
    pts[0].x = p1.x - radius;
    pts[0].y = p1.y - radius;
    pts[1].x = p1.x + radius;
    pts[1].y = p1.y + radius;
    Ellipse::create(pts,2);
  }
 };
```

Notice how simple it was to create a new shape class by using inheritance! There are now only two modifications:

```
Sc.add_shape(4,Circle::New);  // in setup_shapes()
...
// in ShapeWindow::keydown.
case 'C': m_obj = new Circle(); break;
```

This is all quite straightforward; the `ShapeWindow` class must be modified, but `ShapeContainer` needn't be. The object-oriented version is perhaps more work up front, but the payoff is that the resulting program is easy to extend. Figure 8.6 shows the class hiearachy for this case study.

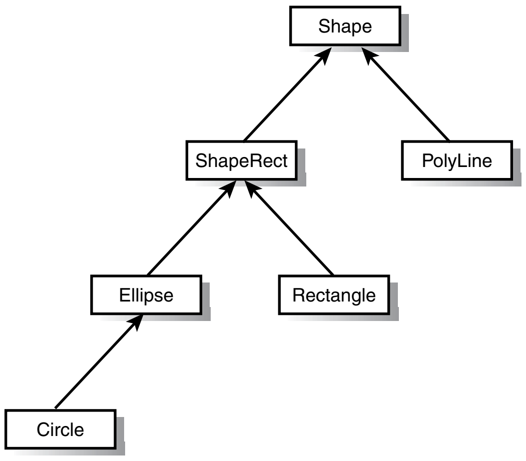


Figure 8.6: *Class hierarchy for the drawing program.*

Some suggestions for further work:

- Introduce color into these shapes, for both the foreground and the background.
- Explain how selection of objects could work.
- Explain what would be needed to have true composite shapes. That is, how can `ShapeContainer` itself be treated as a full-grown **`Shape`**?

What's Next?

At this point, you have started to see object-oriented programming in action. It may seem like a lot of difficult typing at the moment, and I'm sure you can think of other kinds of solutions. Remember that it is important for programs to grow and be extended, and object-oriented programming builds a foundation for that growth. It encourages code reuse on a higher level than does structured programming. It *decomposes* programs into self-sufficient modules that can easily be given to different programmers to finish.

In Chapter 9, "Copying, Initialization, and Assignment," we'll discuss objects in more detail. We will discuss how C++ objects are copied, how to customize type conversions, and generally how to invent fully defined types for which you have full control over every aspect of the object's life.

Copying, Initialization, and Assignment

C++ code can be short, precise, and clear. For example, to append a string to another string, you can use `s1 += s2`, whereas a C programmer would have to use `strcat(p1,p2)`, which is not only less obvious but can be a problem if the buffer `p1` is too small to hold all the characters. Similarly, the simple C++ assignment `s1 = s2` takes care of any copying and reallocation that is necessary. Not everyone who drives cars needs to learn about carburetors, but you are a better car owner if you know that your car contains fuel, water, and oil systems rather than straw and oats. This chapter gives a quick tour under the hood of the C++ object model: what its hidden costs are, and what its weak points are.

In this chapter you will learn

- The difference between initialization and assignment
- What memberwise copying is and why it isn't always appropriate
- How to customize initialization behavior
- About value semantics, as opposed to pointer semantics

Copying

Programmers make certain assumptions about ordinary number variables in programming. One of these is that `x = y` means that the value of `y` is copied to `x`. Thereafter, `x` and `y` are completely independent of one another, and changing `y` does not change `x`. References and pointers do not follow this rule; it is easier to share a reference to an object than to make a copy of that object, but you then can't guarantee that the data will not be changed by someone else. The behavior of ordinary variables is called *value semantics*, where *semantics* refers to the meaning of a statement, as opposed to its *syntax*, which is how you write it.

There are fewer surprises with value semantics than with pointer semantics, but it involves copying and keeping redundant information. For example, you might have a `vector<int> v1` with 1,000,000 elements; the simple assignment `v2 = v1` would involve a lot of copying and use up an extra 4MB of memory. Therefore, understanding how C++ manages copying is crucial to writing good, fast, and reliable code.

Initialization versus Assignment

Two distinct kinds of copying happen in C++, and they are often confused with one another because they both usually use `=`. An object can be *initialized* when it is declared, and thereafter it can be *assigned* a new value. But initialization and assignment are not necessarily the same. This is most obvious in the case of references, such as the following:

```
;> int i = 1, j = 2;
;> int& ri = i;  // initialization
;> ri = j;       // assignment
```

The initialization `ri = i` means "make `ri` refer to `i`," and the assignment `ri = j` means "copy `j` into whatever `ri` is referring to," which means the assignment is actually modifying the number stored in `i`. References are an exceptional case, and for ordinary types, there is no effective difference between initialization and assignment—the results of `int i; i = 1;` and `int i = 1` are the same. But even though the results must be consistent, class initialization and assignment are different operations. Here is a silly example that uses the old C output routine `puts()` to show when the constructors and `operator=` are called:

```
struct A {
  A()                     { puts("A() default"); }
  A(const A& a)           { puts("A() init");    }
  A& operator=(const A& a) { puts("A() assign"); return *this; }
};
;> A a1;
```

```
A() default
;> A a2 = a1;
A() init
;> a2 = a1;
A() assign
```

The declaration `A a2 = a1` must involve a constructor, and it matches the second declaration of `A()` with a `const` reference argument; the assignment `a2 = a1` matches `operator=`. Basically, initialization involves construction and copying, whereas assignment involves just copying. If you leave out the second constructor `A(const A& a)`, C++ will generate one; it will **not** use the assignment operator. If `operator=` is not present, the compiler will do sensible default copying (more about this in the next section.)

Initialization does not need a `=`. C++ is equally happy with object-style initializations, in which the argument is in parentheses. Ordinary types can be initialized with values in parentheses as well, and in fact this is the syntax for constructor initialization lists. So you can rewrite the preceding example as follows:

```
;> A a2(a1);
A() init
;> a2 = a1;
A() assign
;> int k(0);
;> k = 0;
```

The second constructor of `A`, which takes a `const` reference argument, is called a *copy constructor*, and it is used in all initializations. Initialization happens in other places as well. Passing an object to a function by value, as in the following example, is a common case:

```
;> void f(A a) {  }
;> f(a1);
A() init
```

Effectively, the formal argument `a` of `f()` is a local variable, which must be initialized. The call `f(a1)` causes the declaration `A a = a1` to happen, so `f()` receives its own local copy of `a1`. You can also return objects from functions, as in the following example:

```
;> A g() { return a1; }
;> a2 = g();
A() init
A() assign
```

You should try to avoid passing and returning objects for large structures because of the copying overhead. Passing a `const` reference or returning a value via a reference does not cause copying to occur.

Memberwise Copying

What is the default, sensible, way that C++ copies objects? If you consider the following `Person` structure, what must happen to properly copy such objects?

EXAMPLE

```
struct Person {
   string m_name;
   string m_address;
   long   m_id;

   Person()
    : m_name("Joe"),m_address("No Fixed Abode"),m_id(0)
   {}
};
;> Person p;
;> p.m_name = "Jack";
(string&) 'Jack'
;> Person q;
;> p = p;
;> q.m_name;
(string&) 'Jack'
```

People imagine at first that objects are copied byte-by-byte; C copies structures this way and provides the library function `memcpy()` for copying raw memory. However, you should avoid using `memcpy()` on C++ objects; instead, you should let the compiler do the work for you because the compiler knows the exact layout of the objects in memory. For instance, C++ strings contain pointers to dynamically allocate memory, and you should not copy these pointers directly. (You'll learn why this is the case in the next section.) C++ automatically generates the following assignment operator for `Person`:

```
Person& Person::operator= (const Person& p)
{
  m_name = p.m_name;
  m_address = p.m_address;
  m_id = p.m_id;
  return *this;
}
```

This type of copying is called *memberwise copying*: All the member fields of the source are copied to the corresponding fields of the target. Memberwise copying is not the same as merely copying the bytes because some members have their own `=` operators, which must be used. Of course, if a simple structure contained no objects, such as `string`s or containers, memory would in effect simply be moved, and you can trust C++ to handle this case very efficiently. In the same way as `operator=`, C++ generates the following copy constructor, unless you supply your own:

```
Person::Person(const Person& p)
 : m_name(p.m_name), m_address(p.m_address),m_id(p.m_id)
{ }
```

Copy Constructors and Assignment Operators

Why would you need control over copying? C++ memberwise copying generates default *copy constructors* and assignment operators that work well for most cases. But in some cases memberwise copying leads to problems.

The following is a simple Array class, which is an improvement over the one discussed in Chapter 7, "Classes," since its size can be specified when the array is created:

```
class Array {
 int *m_ptr;
 int m_size;
public:
 Array(int sz)
 {
   m_size = sz;
   m_ptr = new int[sz];
 }
 ~Array()
 {
   delete[] m_ptr;
 }
 int& operator[] (int j)
 {
   return m_ptr[j];
 }
};
```

This is the simplest possible dynamic array; space for sz integers is dynamically allocated by new int[sz]. This array form of new makes room for a number of objects of the given type, which are properly constructed if they are not simple types. (To ensure that all these objects are destroyed properly, you need to use delete[]. This will not make any difference for int objects, but it's a good practice to always use it with the array new.) The m_ptr pointer can then be safely indexed from 0 to sz-1; you can easily put a range check in operator[]. To prevent memory leaks, you should give the memory back when the array is destroyed; hence, you use delete[] m_ptr in the destructor. Here is the Array class in action:

```
;> Array ar(10);
;> for(int i = 0; i < 10; i++) ar[i] = i;
;> Array br(1);
;> br = ar;
```

```
(Array&) Array {}
;> br[2];
(int&) 2
;> ar[2];
(int&) 2
;> br[2] = 3;
(int&) 3
;> ar[2];
(int&) 3
```

Everything goes fine until you realize that you are not getting a true copy of `br` by using `br = ar`; the second array is effectively just an alias for the first. This is a consequence of memberwise copying; `m_size` and `m_ptr` are copied directly, so `br` shares `ar`'s block of allocated memory. This is not how value semantics works, and it can be confusing and cause errors because you are working with the same data, using two different names. In precisely the same way, initialization will be incorrect. That is, `Array b = a` will cause `b` to share the same alloated block of memory as `a`. Figure 9.1 shows the situation; `ar.m_ptr` and `ar.m_ptr` are the same!

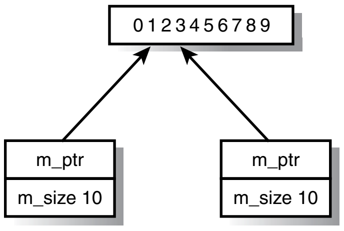


Figure 9.1: *`br` and `ar` refer to the same memory block.*

You will probably find out sooner rather than later; the following simple test function crashes badly when the arrays go out of scope and are destroyed:

```
void test()
{
  Array a(10),b(10);
  b = a;
}
```

After the assignment, both the `a` and `b` have a pointer to the same block of memory. The pointer is given back to the system when `b` is destroyed (which calls `delete m_ptr`.) After a pointer is deallocated, you should have nothing to do with it, and you should especially not try to dispose of it again, which is what happens when `a` is destroyed.

So in these cases it is necessary to explicitly define copy constructors and assignment operators. They essentially do the same thing: They both call `copy()`, which allocates the pointer and copies the values from another `Array`; it can be defined like this:

EXAMPLE

```
void Array::copy(const Array& ar)
{
  m_size = ar.m_size;
  m_ptr = new int[m_size];

  int *ptr = ar.m_ptr;
  for(int j = 0; j < m_size; j++) m_ptr[j] = ptr[j];
}
Array::Array(const Array& ar)
{
  copy(ar);
}
Array& Array::operator= (const Array& ar)
{
  delete m_ptr;
  copy(ar);
  return *this;
}
```

Factoring out the copy code into `Copy()` means that you can make sure that initialization and assignment are in step with one another. C++ has no way of knowing whether you have defined the initialization and assignment operations consistently. You should always define both operations or neither of them.

Some experts say you should always supply the initialization and assignment operations explicitly like this, so that it is clear how the class handles copying. Stanley Lippman (in a Dr Dobbs Journal article) pointed out that for simple classes that have to be very efficient, it's best to let the compiler handle copying the code. In such cases, you should be sure to include a comment which states that you have left out the initialization and assignment operations.

Copying with Inheritance

You can understand inheritance best if you consider the base class to be a member object. That is, if you inherit `A` from `B`, then `B` contains an `A` object, as in the case of `Person` containing `Employee` in Chapter 8, "Inheritance and Virtual Methods." `B` usually inherits all of `A`'s members, except for constructors, destructors, and the assignment operator. This is as it should be; `B` is usually a different beast from `A`, with extra data, and using the old assignment leaves the extra fields of `B` uninitialized. But if you don't supply copy constructors or assignment operators, then the compiler does memberwise copying on B's fields, and it uses the inherited methods to initialize the base class.

```
struct obj {
  obj()             { puts("obj() default"); }
  obj(const obj& o) { puts("obj() init");    }
};
struct A {
  obj mem_a;
  A()               { puts("A() default"); }
  A(const A& a)     { puts("A() init");    }
};
struct B: A {
  obj mem_b;
  B()               { puts("B() default"); }
};
;> B b1;
A() default
B() default
;> B b2 = b1;
obj() default
A() init
obj() init
```

As expected, the default constructor for B first calls the default constructor for A. Remember the basic rule in operation here: C++ guarantees that everything in an object will be properly initialized, including the base class. There is no explicit copy constructor for B, so the compiler generates a copy constructor that does memberwise initialization. You can think of the base class A as the first member; it is initialized first, which causes A's copy constructor to be called. The member object mem_b is properly initialized, but only the default constructor is called for mem_a.

If you supply a copy constructor, you should not assume that the inherited copy operation will take place. In the following example, B has a copy constructor, but only A's default constructor is called:

```
struct B: A {
  obj mem_b;
  B(const B& b)  { puts("B() init"); }
};
;> B b1;
;> B b2 = b1;
A() default
B() init
```

The default base constructor is called when a class is constructed and there is no explicit call in the class initialization list. You might be tempted to do the base initialization explicitly, but it is a better idea to rely on the official copy constructor, which is called on if you ask nicely, as in the following example:

```
struct B: A {
   obj mem_b;
   B(const B& b)
    : A(b)
    { puts("B() init"); }
 };
;> B b2 = b1
A() init
B() init
```

Similarly, if you supply `operator=`, you should be prepared to call the inherited operator explicitly, as in the following example:

EXAMPLE

```
struct A {
  A& operator=(const A& a) {
    puts("A() =");
    return *this;
  };
};

struct B: A {
  B& operator=(const B& a)
  {
    A::operator=(a);   // call A's copy assignment operator directly!
    puts("B() op=");
    return *this;
  }
};
;> B b1,b2;
;> b1 = b2;
A() =
B() =
```

In this example, `A::operator=(a)` is merely calling the operator = directly as a function. You can do this with all operators; for instance, `operator+(s1,s2)` is the same as `s1+s2`. You have to call the operator like a function to specify that the inherited operator is needed. You cannot use `*this = a`, even though it compiles, because it finds `B::operator=`, and the program will crash quickly due to uncontrolled recursion. Alternatively, if you have defined `A::operator=` by using some `copy()` method (as recommended earlier in the chapter), you can call that directly.

The Life of Objects

Everybody gets a little confused at first, when learning about C++ copy semantics, so don't worry if your head hurts a bit. An educator noted at a recent American Academy of Sciences meeting that the average soap opera is much more complicated than most mathematics. The C++ object model is certainly more straightforward than *The Bold and the Beautiful* (but then I probably don't watch enough TV). There are basically two ways to deal with objects:

- Dynamically creating and destroying objects
- Automatically creating and destroying objects

Dynamic Creation of Objects

You can create objects dynamically and use references or pointers to refer to them. In this case, assignment results in another alias to the object, and if you want a genuine copy, you have to ask for it explicitly. Generally, this is how Delphi and Java prefer to do things, and of course you can do this as well in C++. The following shows a `String` class (which is different from the standard one), with the various operations expressed as methods:

```
;> String *ps1 = new String("hello");
;> String *ps2 = new String("polly");
;> String *ps3 = ps2;
;> *ps2 = "dolly";      // changes *ps2 as well!
;> ps3 = ps2->clone();  // now ps3 is independent
;> cout << ps1->append(ps2) << endl;
hellodolly
```

Initially `ps2` and `ps3` point to the same object, and only after you explicitly call `String::clone()` are they independent. This style seems awkward if you are used to normal C++ strings. In fact, both Delphi and Java regard strings as exceptional objects and overload the + operator to mean 'concatenate string.' The advantage of this style is that everything is out in the open: Construction and copying are explicit, and (except in Java) destruction is also explicit. So you have full control of the life of the objects. It is also efficient because you are passing references around and not copying excessively.

The difficulty is twofold: First, the lack of "semantic sugar" (that is, operator overloading) can make code look awkward, and second, there are always problems with object lifetime. If you delete an object prematurely, pointers scattered throughout the system might still refer to it. Any attempt to access these dead objects leads to trouble, usually access violations (it's also common to try to delete objects twice). Even worse, the system might have reallocated that space to another object of the same type, in which case you are really in trouble because the program is then subtly wrong. On the other hand, if you don't delete objects, the free memory is eventually exhausted.

Java's solution to this problem is similar to how modern society consumes things: It assumes that resources are infinite and hopes that recycling will save the day. Occasionally, the system runs the garbage collector, which identifies objects that are no longer in use. (Garbage collection, by the way, is not restricted to Java. Some large C++ programs rely on it as well—it is a technique for memory management and does not require direct language support.) The argument for using garbage collection is that people cannot be relied on to dispose of their own discarded data. It is indeed easy to mismanage the life of dynamic objects, and it is a major cause of unreliability in C++ programs. Several techniques can help, however, and we'll talk about them later in this chapter, in the case study "A Resizable Array."

Interestingly, C++ allows you to change the meanings of `new` and `delete`. They are operators, after all, and most C++ operators can be overridden. Of course, you need to do something sensible with them: They must manage memory allocation. You can pass the request to the usual operators by calling them directly. In the following example, you want to keep tabs on the number of `Blob` objects that have been created:

```
class Blob {
  int m_arr[100];
public:
  int *ptr()  { return m_arr; }

  void *operator new (size_t sz) {
     kount++;
     return ::operator new(sz);
  }
  void operator delete (void *ptr, size_t sz) {
     kount—;
     ::operator delete(ptr,sz);
  }
};
```

Any dynamic allocation of `Blob` uses `Blob::operator new`, and any disposal of a `Blob` object with `delete` uses `Blob::operator delete`. So, you can have complete control over dynamic allocation; in this case, the overloaded operators call the system versions using the global scope operator (`::`). Dynamic allocation can be an expensive operation, and custom solutions can speed things up dramatically. Another use of this would be to use this technique to switch off the disposal of memory completely for a particular class.

As with any advanced C++ technique, remember that you should only do this if you have a very good reason—and proving that you can overload `operator new` is not good enough. In years of working with C++, I've only **had** to really do this twice.

Automatic Destruction of Objects

The second approach to dealing with objects involves declaring objects directly and letting C++ automatically dispose of them at the end of their lives. C++ guarantees that local automatic (that is, non-`static`) variables are automatically destroyed, no matter how you left the function. This makes C++ exception handling well behaved. Ordinary global variables are destroyed when the program closes. They are also created before `main()` is called, so you could use the following code to run initialization code for a module:

```
struct _InitMOD1 {
   _InitMOD1() {
      puts("MOD1 initialized");
    }
  ~_InitMOD1() {
      puts("MOD1 finalized");
    }
 };
InitMOD1 _InitVar;
```

This might look a bit ugly, but the preprocessor can make it quite elegant.

A class member variable is destroyed when its object is destroyed. The compiler always generates a suitable destructor for any objects that themselves contain objects. Of course, this does not apply to pointers in objects, which need to be explicitly destroyed in destructors.

Temporary Objects

It is possible for local objects to be nameless. These nameless objects only exist for the duration of a statement. For example, the following code shows a shortcut for writing out to a file, together with an equivalent long-hand version:

```
;> ofstream("out.txt") << "count is " << n;
;> { ofstream tmp("out.txt"); tmp << "count is " << n; }
```

The first line of this example opens a file, writes a string and an integer to the file, and closes the file. The second line of this example is equivalent code that shows what actually happens when you run the first line: A *temporary object* is created, used, and destroyed. A temporary local context around the statement forces the temporary object to go out of scope.

Temporary objects are created all the time, and usually you don't need to know about them, but they are an essential part of the C++ object model. For example, consider these `string` operations:

```
;> string s3 = s.substr(0,3);
;> string s3s = s3 + s;
```

The method `string::substr()` constructs and returns a temporary string object, which is used to initialize `s3`. The temporary object is then destroyed. Likewise, `s3+s` returns a temporary object. The generation of unnecessary temporary objects can slow down operations considerably. In the following code, the first expression `s += s3` is much faster than `s = s + s3`, (especially if the strings are particularly big) because no temporaries are created. I have written out the last expression in full, to show that it involves the creation of a temporary, the concatenation, and two copies.

```
;> s += s3;
(string) 'hellohel'
;> s = s + s3;
(string) 'hellohelhel'
;> { string tmp = s; s += s3; s = tmp; }  // what actually happened in 's = s + s3'
```

Beware of keeping references to temporary objects. Here are a few problem areas:

```
;> const char *p = (s1+s2).c_str();
;> cout << p << endl;   // can be utter garbage!
;> string& f() { return s.substr(0,3); }
```

The pointer returned from `c_str()` should never be kept because it's generally valid only as long as the string lasts; in this case, `(s1+s2)` is a temporary object. This is an example of the trouble that comes with mixing high-level objects with low-level code. The function `f()` shows why you should be careful about returning references. Any references to a local variable—or a temporary variable, in this case—will be a source of trouble if that variable is out of scope.

You can use a constructor of a class as you use a function. Remember that you originally defined `make_point()` to create `Point` objects. It is possible to use `Point(x,y)` in expressions in a similar way to `make_point(x,y)`:

```
;> Point p = Point(x,y);  // legal, but silly
;> int get_x(Point p)
;>  { return p.x; }
;> get_x(Point(100,200));
(int) 100
```

Using constructors in this way creates temporary objects as well, so you don't have to declare a `Point` variable explicitly and pass it as an argument.

By now you have probably seen a lot of functions that take `std::string` arguments, and you have often called them with string literals (for example, `str2int("23")`). However, you do not have to explicitly call the string constructor (that is, you do not have to use `str2int(string("23"))`) because C++ automatically uses the constructor for converting `char *` into `std::string`. This is not a special case; you can control the type conversion of any of your classes, which is the subject of the next section.

Type Conversion

Type conversion is an important part of the behavior of any type. The usual conversion rules related to numbers and pointers are called the *standard conversions*. For instance, any number will be converted to a double floating-point number, if required; any pointer will convert to `void *`, and so on.

C++ gives the designer of a class full control over how that type behaves in every situation, and so you can specify how other types can be converted into the type. This is set up for you if you have defined constructors taking one argument. For instance, the constructor `string(char *)` will be used by the compiler to convert `char *` into `string`. This saves a lot of typing because you can then pass string literals (that is, text in quotes) to functions expecting a `string` argument. If there was a constructor `string(int)` as well, then you could also pass integers to such functions. (This would probably not be a good idea; the more "silent" conversions that are possible, the more likely you are going to be unpleasantly surprised.)

But remember the cost: Every silent conversion using a constructor causes a temporary object to be created. If you need serious speed, you can always overload a function to achieve the same result—that is, define another function that takes a `char *` value directly. For instance, the following example is one way to make `str2int()` faster; the first function supplies another way to call the C library function `atoi()`, and the second function passes the `string`'s character data to the first function:

```
inline int str2int(const char *s)
   { return atoi(s); }
inline int str2int(const string& str)
 { return str2int(str.c_str()); }
```

The `inline` function attribute explicitly asks the compiler to insert the function's code directly into the program wherever the function is called, so there is no extra cost in giving `atoi()` a new name and identity as `str2int()`. But you should test a program before going on a mad drive for maximum efficiency, and ask yourself if shaving off 50 milliseconds is going to affect the quality of your users' life.

NOTE

`const` in the first function's argument is very important. I was bitten by this recently, so you should share my experience, if not my pain. If the character pointer is not `const`, then `str2int(str.c_str())` cannot match the first signature because `str.c_str()` returns `const char *` and C++ will never violate "constness" by converting `char *` to `const char *`. Instead, it matches the other signature by type conversion to `const string& s`. The snake proceeds to eat its own tail, and the recursion will end only when the program crashes.

The important thing to note with these two functions is that the compiler does not try to force a conversion of character literals because there is already a function that matches them perfectly well. User-defined conversion is only attempted if there is no other way to get a function match.

Note that user-defined conversions also apply to operators, which after all are just a fancy form of function call. If you were using a string class that has only defined `operator==(const string&, const string&)`, you would still be able to compare string literals, as in `name=="fred"` or `"dog"==animal`, because the `string(char *)` constructor is used to convert those literals. But the compiler will need to generate temporary `string` objects, and you might find that a fast string comparison becomes surprisingly slow. Again, the solution is to overload `operator==`, at least for the first case. (The second case isn't as eccentric as it seems; some people write comparisons like this so they won't be bitten by `name="fred"`.) Remember that assignments are also operator calls, and C++ strives to convert the right-hand side into the left-hand side by means of user conversions.

Sometimes you simply don't want an automatic conversion. In the past, people would prevent automatic conversion by using clever class design. For example, there is no `std::string` constructor that takes a single integer argument. The idea would be that you could generate a blank string of `n` characters with `string(n)`, but it would have strange consequences. The call `str2int(23)` would translate as `str2int(string(23))`, which ends up returning zero because `atoi()` is so tolerant of spaces and non-numeric characters. `str2int(23)` is definitely a misunderstanding or a typo and should cause an error. So the `string` constructor is designed so that you must use `string s(8,' ')` to get an empty string `s` with eight spaces.

Standard C++ introduces a new keyword, `explicit`, which tells the compiler not to use a constructor in such conversions. For example, the standard vector class has the following constructor:

```
explicit vector(size_type sz = 0);
```

`explicit` before the declaration of this `vector` constructor allows you to declare `vectors` such as `vector<int> vi(20)` without getting puzzling attempts to convert integers into vectors.

So far you have seen how to control how C++ converts other types to your class. The other kind of conversion operation involves converting your class into other types. For example, say you have a `Date` class that has an `as_str()` method for returning the date in some standard format. Then the following *user-defined conversion operator* in the class definition causes `Date` objects to freely convert themselves into `strings`:

```
operator string () { return as_str(); }
```

If C++ tries to match any function argument with `Date` objects and can't find any obvious match, it leaps at the chance to convert the date into a `string`. It would probably not be a good idea for a `Date` object to want to convert itself into an integer, both because that would not be unique (does it refer to a Julian date, a star date, or the number of seconds since Halloween?) and because integers are too common as function arguments.

You can get surprising control over an object's behaviour by using user-defined conversions. The following code uses YAWL, which was mentioned in the last chapter in the section "Class Frameworks." (See Appendix B, "A Short Library Reference" for more on YAWL.) `TWin` is a type that describes a window, and it has a static member function `get_active_window()` that returns the the window which is currently active. When using the UnderC interactive prompt, this is the console window itself. `TWin` has two methods `get_text()` and `set_text()` for accessing a window's caption (that is, text in the title bar):

```
;> #include <yawl.h>
;> TWin *w = TWin::get_active_window();
;> w->get_text();
(char *) "UnderC for Windows"
;> w->set_text("hello, world!");
```

This is a classic pair of get and set methods. Borland's C++ Builder has a non-standard C++ extension called a *property*. Properties look like simple variables, but they have different meanings depending on which side of an assignment they appear. The preceding YAWL example would be written like this in C++ Builder:

```
  String s = w->Caption;         // same as get_text() above
  w->Caption = "Hello, World!";  // same as set_text() above
```

`w->Caption` looks like a straightforward member variable access but is actually an action. `w->Caption = "Hello, World!"` not only sets a value, but updates the window's caption. User-defined conversions make this possible using standard C++ as well. Consider the following class:

EXAMPLE

```
class TextProperty {
  TWin *m_win;
public:
  TextProperty(TWin *win) : m_win(win) {}

  void operator=(const string& s)
    { m_win->set_text(s.c_str()); }

  operator string ()
    { return m_win->get_text(); }
};
;> TextProperty text(TWin::get_active_window());
;> string t = text;      //TextProperty::operator string()
;> text = "hello dolly"; //TextProperty::operator=(const string&)
```

In the initialization `string t = text`, the only way to match a `TextProperty` value to a `string` value is to convert the `text` object into a `string` using the user-defined conversion. This will also happen with any function expecting a `string` argument. The `string` conversion operator then actually gets the window text.

If the object `text` appears on the left-hand side of an assignment, then its meaning changes completely. It will then match `operator=(const string& s)`, which has the effect of setting the window text. This is precisely how properties are meant to work; what looks like the simple use of a variable causes either get or set code to be executed. In the first case study, you will see how this technique can make an intelligent array possible.

Interestingly, it was once common for C++ string classes to automatically convert to `const char *`; the Microsoft Foundation Class `CString` class behaves like this. But it led to too many odd surprises, and so the standard string has the `c_str()` method to do this explicitly. The problem is that C++ gives you the power to design your own language, and you should use this power wisely or not at all. User-defined conversion operators are like the color purple with amateur Web designers: They are best avoided until you know what you're doing.

Case Study: A Resizable Array

It can be a useful exercise to reinvent the wheel, especially if you don't insist on using the results. `std::vector` and `std::valarray` (see Appendix B, "A Short Library Reference") will do well for most applications and have been carefully designed. Also, by now, C++ programmers know them and may be irritated if they have to learn new classes that you have created to do the same thing as `std::vector` and `std::valarray`.

But the issues of copying and value semantics are best illustrated by something nontrivial. Furthermore, the resizable array discussed in this section shows how you can have objects that obey value semantics that are not expensive to use.

As mentioned previously in this chapter, you should always pass complex objects by reference because of the copying overhead involved. (Something small and nimble, such as the `Point` class, can probably be moved as fast as a reference.) But then in the examples in this book, we often pass a `std::string` argument by value; isn't that inconsistent? If you really need string manipulation to be fast, you would be better off with passing `const string&` arguments, but the cost of passing by value is often not as great as you think.

You want to pretend as much as possible that `string` is part of the language's original furniture, like `int` or `double`, and for that value semantics must not be too expensive. One solution is a sharing scheme, in which a number of string objects actually share the same data. So passing by value is just passing an alias, as in the case of the array class defined earlier in this chapter, in the section "Copy Constructors and Assignment."

But as soon as someone tries to write to this string, the string creates its own unique copy of the data. Thereafter it can relax. This is called *copy-on-write* and can save a lot of pointless copying. (It resembles modern manufacturing practice, where a computer manufacturer will assemble the item when it's ordered, to minimize inventory.)

How do the strings know that they're sharing data? Because the data is *reference counted*. That is, the data has an associated count, starting at 1. Anybody who wants to borrow the data must increment the reference count. Anybody who no longer needs the data must ask the data nicely to dispose of itself, in order to decrement the reference count. When the reference count becomes zero, the data knows it's no longer needed and commits hara-kiri. Here is a base class that implements this strategy:

EXAMPLE

```
// C++ By Example, Chapter 9
// ref_count.h
class RefCount {
   int m_ref_count;
public:
   void init_ref_count() { m_ref_count = 1; }
   void incr_ref()       { ++m_ref_count; }
   void decr_ref()      { —m_ref_count; }
   int  ref_count()     { return m_ref_count; }

   RefCount()           { init_ref_count(); }
   virtual ~RefCount()  { }

   virtual void dispose()
   {
     —m_ref_count;
       if (m_ref_count == 0) delete this;
   }
};
;> RefCount *p = new RefCount();
;> p->incr_ref();    // i.e. now shared by two objects
;> p->dispose();     // only by one!
;> p->dispose();     // finally gets deleted
```

The basic idea of reference-counted objects like `RefCount` is that the reference count is the number of owners of that object. Any would-be owners must cooperate in two ways: They must call `incr_ref()` when taking possession of a reference, and they must call `dispose()` when they don't need that reference anymore. It is more accurate to say that such objects are never owned, but only borrowed.

Please note the *virtual destructor* for `RefCount`. Like any method, a class's destructor can be overridden, and in fact it is recommended practice in any hierarchy involving polymorphic methods. You need to make the destructor virtual here because the `dispose()` method must call the correct destructor when it issues the suicidal command `delete this`.

The new `Array` class simply has a pointer to the actual representation object. Copying such arrays just copies these pointers, and it also increments the representation's reference count. So the first task is to work on the representation, which in the preceding example is a `vector`-like class that has the familiar `std::vector` interface.

A Resizable, Reference-Counted Vector

The implementation of the `Vector` class uses two pointers to `int`, called `m_begin` and `m_end`, which correspond directly to the `begin()` and `end()` methods (that is, `m_end` points just past the end of the allocated block pointed to by `m_begin`). It is not necessary to keep the number of elements as a member variable, but you can anyway for efficient access. The class uses `typedef` to make `T` into an alias for `int` so that you can easily use the same code for other types, simply by changing the `typedef`.

A simple pointer works fine as an iterator (after all, iterators are generalized pointers)—`*p` accesses the value, and `++p` accesses the next value. This example shows how iterators and pointers behave in very similar ways. Because the size of the type `int` is 4 bytes, a pointer increment actually adds 4 bytes to the address each time, as you can see from the value of `++pi`:

```
;> int *pi = new int[10];
;> for(int i = 0; i < 10; i++) pi[i] = i;
;> pi;
(int*) pi = 8F1E80
;> *pi;
(int) 0
;> ++pi;
(int*) 8F1E84
;> *pi;
(int) 1
;>  –pi;
(int*) 8F1E80
;> int *p_end = pi + 10;
;> for(int *p = pi; p != p_end; ++p)
;1} cout << *p << ' ';
0 1 2 3 4 5 6 7 8 9
```

The first part of the interface to `Vector` looks like this. `iterator` is defined as a `typedef`; this type will be accessed as `Vector::iterator`, precisely like with the standard containers. A *nested* class `RangeError` is defined as a type to be thrown when there's an 'index out of bounds' error, which will simularly be refered to as `Vector::RangeError`:

```
class Vector: public RefCount {
 public:
  typedef int   T;
 private:
   T *m_begin;
   T *m_end;
   int m_size;
   int m_cap;
 public:
   typedef T *       iterator;

   struct RangeError {
     int idx;
     RangeError(int i) { idx = i; }
     int where() { return idx; }
   };
   iterator begin()  { return m_begin; }
   iterator end()    { return m_end;   }
   int size()        { return m_size; }
   int capacity()    { return m_cap;  }

   T& operator[] (int i)
   { return m_begin[i]; }

   T& at(int i)  {
    if (i < 0 || i >= m_size) throw RangeError(i);
    return m_begin[i];
   }
```

You know that the iterator is really just an `int*`, but you would like to keep this implementation detail to yourself. (It does not have to be a pointer and will certainly not be a plain pointer in the case of a linked list.) As with `vector`, the `at()` method provides checked access, and `operator[]` provides unchecked, fast access.

As well as the current size `m_size`, you also keep the actual capacity of the array `m_cap` because, for efficient operation, an array must overallocate. That is, the actual size of the allocated block is larger than the number of elements. In the following allocation code, the basic operation is `_realloc()`, with which `_alloc_copy()` and `_grow()` are defined:

```
void _copy(T *start, T *finish, T *out)
{ while (start != finish) *out++ = *start++; }

void _realloc(int new_cap, int new_sz, const Vector& v)
{
   T *tmp = new T[new_cap];
   if (m_begin) {
     _copy(v.m_begin,v.m_end,tmp);
      delete[]m_begin;
   }
   m_begin = tmp;
   m_end = m_begin + new_sz;
   m_size = new_sz;
   m_cap  = new_cap;
 }

void _alloc_copy(const Vector& v)
{ realloc(v.m_cap,v.m_size,v); }

void _grow()
{ realloc(m_size+NGROW,m_size,*this); }
```

By using `_alloc_copy()`, you can define the copy constructor and copy assignment operators, as in the following code:

```
Vector(const Vector& v)
{
  m_size = 0;
  m_begin = NULL;
  _alloc_copy(v);
}

Vector& operator= (const Vector& v)
{
  _alloc_copy(v);
  return *this;
}
```

The resizing operations then become simple:

```
void resize(int sz)
{ _realloc(sz+NGROW,sz,*this);  }

void reserve(int sz)
{ _realloc(sz,m_size,*this); }
```

The rest of the `vector`-like interface is as follows:

```
void clear()
 {
    delete m_begin;
    m_size = 0;
    m_cap = 0;
 }

 ~Vector() { clear(); }

 void push_back(T t)
 {
    if (m_size+1 > m_cap) _grow();
    m_size++;
    *m_end++ = t;
 }

 void pop_back()   { m_end—;  m_size—; }
 T back()          { return *(m_end-1); }
 T front()         { return *m_begin; }

};
```

Overallocation makes operations such as `push_back()` much more efficient: If there isn't enough capacity for an extra element, then more space is allocated using `grow()`. As you can see in the following example, the new `Vector` class works almost exactly like `vector<int>`. Note that a `Vector` of 10 elements has room for 20 elements with this implementation:

```
;> Vector v(10);
;> for(int i = 0; i < 10; i++) v[i] = i+1;
;> v.front();  v.back();
(int) 1
(int) 10
;> v.push_back(11);
;> v.size();
(int) 11
;> v.capacity();
(int) 20
;> Vector::iterator vi;
;> for(vi = v.begin(); vi != v.end(); ++vi)
;1}  cout << *vi << ' ';
1 2 3 4 5 6 7 8 9 10 11 ;>
```

The `Array` Class

The implementation of the `Array` class is a pointer to a `Vector` object. So the first part of `Array` is pretty boring—in it, most of `Array`'s methods are delegated to the representation object:

```
class Array {
private:
   Vector *m_rep;

public:
 typedef unsigned int uint;
 typedef Vector::iterator iterator;
 typedef Vector::T T;

iterator begin()      { return m_rep->begin(); }
iterator end()        { return m_rep->end();   }
void clear()          { m_rep->clear();        }
uint size()           { return m_rep->size();  }
uint capacity()       { return m_rep->capacity();  }
void resize (uint sz) { m_rep->resize(sz);     }
void reserve(uint sz) { m_rep->reserve(sz);    }
int  ref_count()      { return m_rep->ref_count(); }

void pop_back()       { m_rep->pop_back();    }
T back()              { return m_rep->back(); }
void push_back(T t)   { m_rep->push_back(t);  }
```

This may look very similar to the situation in the last chapter, where Employee had a Person member variable, and many of Employee's methods simply used the Person methods. There inheritance proved a better solution and saved much typing. However, note that the representation object is not fixed. The whole point of the scheme is that Array contains an indirect reference to Vector.

Here is a more interesting part of Array, where we share those references with other Array objects:

```
// 'copying'!
void attach(const Array& a)
{
  m_rep = a.m_rep;
  m_rep->incr_ref();
}

Array(const Array& a)
{ attach(a);  }

Array& operator= (const Array& a)
{ attach(a);  return *this; }

~Array()
{ m_rep->dispose(); }

Array(int sz = 0)
{  m_rep = new Vector(sz);  }

void unique() {
 if (m_rep->ref_count() > 1) {
    m_rep->dispose();
    m_rep = new Vector(*m_rep);
 }
}

T get(uint i) const
 { return m_rep->at(i); }

void put(int val, uint i)
 { unique(); m_rep->at(i) = val; }
```

Note that Array objects are never copied at first. After a1=a2, a1 and a2 share the same representation object, but the reference count of that object is then 2. Consider this common situation:

```
;> int array_test(Array a, int idx) { return a.get(idx); }
;> Array arr(20);
;> fill(arr.begin(),arr.end(),0);  // initialize all to zero..
;> array_test(arr,2);
(int) 0
;> Array a2 = arr;
;> a2.put(1,0);
```

As usual, passing an object by value causes the copy constructor to be called. That initializes the local variable `a` in the method `array_test()`, and `a` is then destroyed when it goes out of scope. But the initialization does not involve copying, and in this case, the destruction does not involve deallocation. It just decrements the reference count of the shared vector object. This is why you must call `m_rep->dispose()`, rather than `delete m_rep`; `m_rep` will know when it needs to destroy itself.

Similarly, `Array a2 = arr` does not generate a copy, but the `put()` method calls `unique()`, which sees that the representation is shared because the reference count is greater than 1. If the representation object is shared, `unique()` lets go of the shared object and creates a new one by using the `Vector` copy constructor. Figure 9.2 shows the situation: Originally `a` and `arr` point to the same `Vector` object, which therefore has a reference count of 2. A new `Vector` object is created, and the reference count of the first `Vector` object is decremented to 1. The two objects `a` and `arr` are now completely independent.

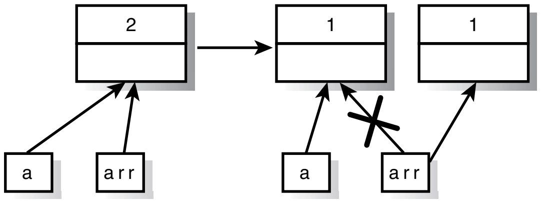


Figure 9.2: *Effect of `arr.unique()`; the objects `a` and `arr` no longer share the same `Vector`.*

Most people would be dissatisfied with an array class that did not overload `operator[]`. Using `get()` to read values and `put()` to write values to the object seems inconvenient. The problem is that C++ can't really distinguish between being on the left- and the right-hand side of an assignment. The technique used to produce a text property in the section on user-defined conversions is useful here. `operator[]` returns a "smart reference" (`IRef`), which has a different meaning on the left-hand side. Here is the helper class `IRef` and how it is used by `operator[]`:

```
struct IRef {
  Array& m_arr;
  uint   m_idx;
  IRef(Array& arr, uint idx)
  : m_arr(arr),m_idx(idx) { }

  void operator= (int val) { m_arr.put(val,m_idx);  }
  operator int   ()        { return m_arr.get(m_idx); }
};

IRef Array::operator[](uint i)
{ return IRef(*this,i); }
```

If an IRef object is found within an integer expression, the compiler will try to convert it to an `int`. The user-defined conversion operator will then actually get the value by using the `Array::get()` method.

If the IRef is on the left-hand side of an integer assignment, the assignment operator (`operator=`) is chosen, which causes the `Array::put()` method to be called. This in turn guarantees that the representation becomes unique.

It seems as though a lot is happening for each simple array access—first the indirect access and then the temporary `IRef`. However, most of the methods involved are easy to inline, and so the runtime cost of the scheme is not excessive.

There are still some things to be done in this example. For instance, support for the operators `+=` and `*=` would need to be put into `IRef` if you wanted to use something like `a[i] += 2`, and `begin()` does not properly check whether the representation is shared. The latter would be useful because it would only need to call `unique()` once in `begin()`, and thereafter, it would produce a very fast pointer access.

Case Study: Writing XML/HTML

NOTE

This case study is a more advanced demonstration of how a collection of cooperating classes (or *class library*) can simplify the generation of HTML. It isn't necessary to know HTML to follow the code, but it will help.

eXtensible Markup Language (XML) has become an important buzzword, and programs are increasingly being expected to be XML compliant. Markup languages should not be confused with programming languages. A markup language is, in fact, all form and very little content. It is a standard way of representing structured data with tags, but the meaning of those tags is not specified. It is more like an alphabet than a language. People get together to decide on a standard for a particular kind of knowledge, such as banking transactions, books, and class hierarchies, and then they give the tags specific meaning. For example, if you needed a more precise way to describe vegetables, you could create Vegetable Markup Language (VEGML), which might look like this:

```
<vegml>
  <root name="potato" size="15" color="brown">
    Boiled Fried Baked
  </root>
  <fruit name="tomato" size="8" color="red">
    Raw Boiled Fried
  </fruit>
  <nut name="acorn"/>
 </vegml>
```

The items in `<>` are usually called *tags* or *elements*, and they can have *attributes*, which are name/value pairs. Between the open tag and the close tag there can be character data. If there is no character data, then you can abbreviate the final tag with a slash (`/`) (for example, `<nut name="acorn"/>`).

A very popular XML-style markup language is Hypertext Markup Language (HTML). Traditional HTML is not quite XML compatible, but it can be made so (for instance, by closing every paragraph tag `<p>` with `</p>`). There are two main differences between HTML and XML: XML tags are case-sensitive and HTML tags are not, and whitespace matters in XML but not in HTML.

```
<html>
<head>
  <title>This is the name of the page</title>
</head>
<body bgcolor="#FFFFFF">
<p>Normal text <a href="anewpage.htm">continues </a>as so.....</p>
<ul>
    <li>here is item 1</li>
    <li>another item</li>
</ul>
<p>Here is <bold>bold</bold> text.
 Here is <italic>italic </italic>text.</p>
</body>
</html>
```

This chapter can't discuss the tags in detail (the excellent NCSA HTML primer at `archive.ncsa.uiuc.edu/General/Internet/WWW/HTMLPrimer.html` is the place to start). In this case study, you will generate some HTML, and a good place to begin is to identify the objects involved: tags, attributes, and documents.

When designing a class library, it is useful to write out how you would imagine it being used. This is easier than first defining the interface for each class. Here is the kind of code I wanted to write: a `HTMLDoc` object is created, a heading "A heading" is written, followed by a new paragraph, and "and some text" in bold.

```
HTMLDoc d("first.xml");
d << head(1) << "A heading" << para << bold << "and some text";
d.close();
```

The idea, of course, is to make our class library work like the `iostream` library. Most of the `iostream` manipulators (such as `width()`; see Appendix B, "A Short Library Reference") that you can put in streams affect only the next field. An alternative is something like `bold(on) << "bold text!" << bold(off)`. But it would be tedious to have to close each tag manually, and that tedium would make it easier to generate badly formed documents. So somehow you have to keep tabs on the tags and close them automatically.

A useful way to think about this problem is to use stacks: You can keep a stack of tags. When a tag is pushed onto the stack, it is opened, and when it is popped, it is closed. For instance, opening the bold tag will put out "<bold>", and closing it will put out "</bold>". So the idea is to pop the stack after putting out text, which closes any pending tags. For instance, after "and some text", the stack contains a bold tag, and popping it will put out "</bold>".

This unfortunately works too well: After `"A heading"`, it closes the heading tag, and it also closes the body and the HTML tags. The solution is to label formatting tags such as headings and bold as temporary tags, and make tags like `<body>` persistent. That is, only temporary tags are popped after an output item. You would finally pop all of the stack, temporary or not, when the document is closed.

`Tag` and `Attrib` classes both have name properties, so you can factor out that part of them as a `NamedObject`. In addition, an `Attrib` object has a value. A `Tag` object can contain a number of `Attrib` objects, but it usually contains none. Any general XML-generating code can be used to generate HTML, and it makes logical sense to consider `HTMLDoc` to be a specialized derived class of `XMLDoc`. `XMLDoc` must have a reference to some open filestream, and it must contain a stack of `Tag` objects.

The following code is the interfaces for the `Attrib` and the `Tag` class (`chap9\attrib.h` and `chap9\tag.h`):

```
class Attrib: public NamedObject {
    string m_value;
public:
  Attrib (string nm="", string vl="")
  : NamedObject(nm), m_value(vl)
  {}
  string value() const { return m_value; }
};

typedef std::list<Attrib *> AttribList;
typedef AttribList::iterator ALI;

// tag.h
#include "attrib.h"
using std::string;

class Tag: public NamedObject {
  bool m_open,m_temp;
  AttribList *m_attribs;
  mutable AttribList::iterator m_iter;
public:
  Tag(string name="", bool temp=true);
  Tag& set(bool on_off);
  void clear();
  bool closed() const       { return ! m_open; }
  bool is_temporary() const { return m_temp; }
  void add_attrib(string a, string v);
  void add_attrib(string a, int v);
  bool    has_attributes() const;
  Attrib *next_attribute() const;
};

typedef std::list<Tag> TagList;
```

Both `Attrib` and `Tag` are derived from `NamedObject`, which you saw in "A Hierarchy of Animals" in Chapter 8. `Attrib` is basically just a name plus a value. As you can see, `Tag` is not a complicated class; it has a name, can be closed or open, and may be temporary. Its main responsibility is looking after a list of attributes. It would have been easy to export attribute list iterators to anybody using this class, but how a `Tag` object organizes its business is its own affair. In particular, other code should not have to make assumptions about the attribute list. So `Tag` has a pair of functions for accessing attributes; the first method, `has_attributes()`, must be tested before you call `next_attribute()`, which returns non-null attribute pointers until there are no attributes left.

In this implementation, `AttribList` is a list of `Attrib *`, and you keep a pointer to `AttribList`, which is usually null. This is the most space-efficient way to implement tags, which often don't have attributes. But these are specific implementation decisions, which could change, and it is the job of `Tag` to keep them to itself. Here is the implementation of `Tag`, based on these assumptions:

EXAMPLE

```
Tag::Tag(string name, bool temp)
   : NamedObject(name),m_open(true),
     m_temp(temp),m_attribs(NULL)
  { }

Tag::Tag& set(bool on_off)
 { m_open = on_off; return *this; }

void Tag::clear()
{
 delete m_attribs;
 m_attribs = NULL;
}

void Tag::add_attrib(string a, string v)
{
    if (m_attribs == NULL) m_attribs = new AttribList;
    m_attribs->push_back(new Attrib(a,v));
}

void Tag::add_attrib(string a, int v)
{
     add_attrib(a,int2str(v));
}

bool Tag::has_attributes() const
{
    if (m_attribs == NULL) return false;
    m_iter = m_attribs->begin();
    return true;
}

Attrib *Tag::next_attribute() const
{
    if (m_iter == m_attribs->end()) return NULL;
    Attrib *a = *m_iter;
    ++m_iter;
    return a;
}
```

Note that `next_attribute()` is a `const` method of `Tag`, which it must be if it is to act on `const Tag&`. But `++m_iter` modifies the object in order to get the next attribute; how can this be allowed? If you look at the declaration of `m_iter`, you will see the new qualifier `mutable`. What we are saying with `mutable` is that modifying the member variable `m_iter` does not really modify the object. It is an iterator to the attribute list, but the attribute list is not itself changed.

You now need to think about the class that does the serious work of generating XML. A list of tags works well as a stack; in fact, most of what `XMLDoc` does is manage this stack, as you can see here:

```
class XMLDoc {
  TagList m_tstack;
  ofstream m_out;
public:
// Tag stack management
  void push_tag(const Tag& tag)  { m_tstack.push_back(tag);  }
  void pop_tag()                 { m_tstack.pop_back();      }
  bool empty()   const           { return m_tstack.size()==0; }
  const Tag& current() const     { return m_tstack.back();    }

  void push(const Tag& tag)
  {
     Attrib *a;
     m_out << '<' << tag.name();
     if (tag.closed()) m_out << '/';
     else if (tag.has_attributes()) {
        m_out << ' ';
        while ((a = tag.next_attribute()) != NULL)
          m_out << a->name() << '='
                << quotes(a->value()) << ' ';
     }
     m_out << '>';
     push_tag(tag);
   }

  bool pop()
  {
    if (empty()) return false;
    Tag& tag = current();
    m_out << "</" << tag.name() << "> ";
    pop_tag();
    return ! empty();
  }
 // streaming out and document management
  virtual void outs(char *str)
  {
    m_out << str;
    while (!empty() && current().is_temporary()) pop() ;
  }

  void outs(const string& s)
  {  outs(s.c_str()); }
```

```
   void open(const string& name)
   { m_out.open(name.c_str()); }

   void close()
   {
   // close out ALL pending tags
    cout << "closing....\n";
    while (pop())   ;
    m_out.close();
   }

   ~XMLDoc()
   { close(); }

 }; // class XMLDoc
```

Notice that all references to `m_tstack` are in the first four methods, which effectively define `XMLDoc` as a stack-like class. The methods `push()` and `pop()` are where the XML specifics are. Because `Tag` is looking after the attributes, this code can run through them simply, without making any assumptions about how `Tag` stores the attributes, using `Tag`'s `has_attributes()` and `next_attribute()` methods.

The `outs()` method is the gateway for all character data; after writing the text to the file stream, it closes any pending tags by popping any temporary tags off the stack. Finally, the document must be opened and closed; when it's closed, the tag stack must be completely emptied with the short and sweet statement `while(pop()) ;`.

Note that `XMLDoc` is not derived from anything. You might be tempted by the thought "`XMLDoc` is a stack of `Tag` objects," and try to inherit from some standard class. This is a bad idea because an `XMLDoc` object **has** a stack of `Tag` objects; it **is not** a stack of `Tag` objects. In particular, inheriting from `list<Tag>` would make `XMLDoc` export all kinds of things that have nothing to do with `XMLDoc`'s job. Likewise, the idea of inheriting from `ofstream` is unwise; you want to force all text data through the narrow gate of the `outs()` method. In this case study, *composition* (that is, building a class out of other classes) makes more sense than inheritance. Inheritance should not be used just because it makes a program seem more object oriented.

To get the intended mode of use, you need to create a few more operator overloads. They may only be syntactic sugar, but these overloads makes creating documents a lot easier. Like `ostream`, `operator<<` is overloaded to output characters strings and `int` values; these all go through the `outs()` method. It is also overloaded for `Tag` arguments; the tag is pushed on the document's tag stack:

```
XMLDoc& operator<<(XMLDoc& doc, char *s)
{
  doc.outs(s);
  return doc;
}

XMLDoc& operator<<(XMLDoc& doc, const string& s)
{
  doc.outs(s);
  return doc;
}

XMLDoc& operator<<(XMLDoc& doc, int val)
{
  doc.outs(int2str(val));
  return doc;
}

XMLDoc& operator<<(XMLDoc& doc, const Tag& tag)
{
   doc.push(tag);
   return doc;
}
```

Up to now you have not seen anything specifically about HTML. But because HMTL is an XML-like language, it is not difficult to specialize `XMLDoc` by creating a derived class called `HTMLDoc`. You should note two things here. First, `XMLDoc` is now a useful part of your toolkit that is available to all other projects you're working on. (There is quite a bit of pressure to make all programs talk to each other in some XML-compatible language.)

Second, `XMLDoc` supplies a concise language that makes the actual job almost straightforward (I say *almost* because nothing in software is trivial). The `open()` method uses the tag-stack interface to generate the tedious bit at the front of all HTML documents. (All this code is found in `chap9\html.cpp`.)

```
void HTMLDoc::open(string name, string doc_title, string clr)
{
  if (name.find(".") == string::npos) name += ".htm";
  XMLDoc::open(name);
  if (doc_title=="") doc_title = name;

  push(Tag("HTML",false));
   push(Tag("HEAD"));
   push(Tag("TITLE"));
   XMLDoc::outs(doc_title);
   XMLDoc::outs("\n");

   Tag body_tag("BODY",false);
   body_tag.add_attrib("bgcolor",clr);
   push(body_tag);
   XMLDoc::outs("\n");
}
```

Note how it is necessary to use the fully qualified name to call the inherited `XMLDoc::open()` method. Here is an example of what `HTMLDoc::open()` generates at the front of the HTML document.

```
<html>
<head>
  <title>This is the name of the page</title>
</head>
<body bgcolor="#FFFFFF">
```

You also have to explicitly use `XMLDoc::outs()` because `HTMLDoc` is going to overload it:

```
void HTMLDoc::outs(char *str)  // override
{
// calls original method to do output
   if (strchr(str,'\n') != NULL) {
     char buff[256];
     strcpy(buff,str);
    for(char *t = strtok(buff,"\n");
     t != NULL; t = strtok(NULL,"\n"))
     {
      push(para);
      XMLDoc::outs(t);
      XMLDoc::outs("\n");
    }
  } else XMLDoc::outs(str);
}
```

Here is some old-fashioned C-style code for a change. HTML text runs together unless you put out paragraph tags. If the character string contains `\n`, then it must be broken up and separated by using paragraph tags. The `strchr()` function returns a pointer to the first match of the specified character; otherwise, it is NULL.

`strtok()` is fairly eccentric: You give it a set of characters (in this case, `"\n"`), which is used to break up the string into tokens. `strtok()` is passed the character string for the **first call**; thereafter it is passed NULL. `strtok()` modifies the buffer we give it; hence the `strcpy()`. (See Appendix B, “A

Short Library Reference," for more information about `strtok()` and other C string functions.)

You still have to define some HTML-specific tags. Here are a few of the most important ones. Simple formating tags work directly (for instance, `<bold>Some text</bold>`), but some tags take parameters. Instead of defining a number of heading tags (`<H1>`, `<H2>`, and so forth) I've defined a function `head()` that modifies the name of the global `head_tag` variable and returns a reference to it. The function `link()` is passed an Internet address, and returns the global `link_tag`. The address is the value of the HREF attribute, which is added to `link_tag`.

```
Tag bold("BOLD");
Tag italic("ITALIC");
Tag link_tag("A");
Tag para("P");
Tag head_tag("");

Tag& head(int level = 0)
{
   head_tag.name("H" + int2str(level));
   return head_tag.set(level > 0);
}

Tag& link(string fname)
{
   link_tag.clear();
   link_tag.add_attrib("HREF",fname);
   return link_tag;
}
```

And finally, here is some exercise code for HTMLDoc:

```
void exercise()
{
  int n = 58;
  HTMLDoc doc("first","An introduction");
  doc << head(1) << "Here is a Title";
  doc << "There have been " << n << " visitors\n";
  doc << bold << "bold text\n";
  doc.close();
}
```

What's Next

By now, you should have a good idea about what happens behind the scenes in C++. You are slightly at the mercy of the system until you appreciate what C++ does for you silently and why it is sometimes wrongheaded. One good way to get a feeling for what C++ does is to single-step through you first C++ programs with a debugger, looking at all the functions you went through. It's a good idea to play with C++ to learn more about it, and UnderC makes that less painful. You can learn a lot by tracing all function calls. In UnderC, you do this as follows, by using the `#opt` command to switch on `t` (for trace) and `v` (for verbose):

```
;> #opt t+ v+
;> s = "hello";
*sig:   void operator=(char*)
*match: string& operator=(char*)
*TRACE <temp>
*TRACE =
*TRACE copy
*TRACE strlen
*TRACE resize
*TRACE _new_vect
*TRACE strcpy
*TRACE strcpy
(string&) 'hello'
;> #opt t- v-
```

If you repeat this with `string s = "hello"`, you will indeed see that `__C__` (that is, the constructor) is called instead of `operator=`.

Chapter 10, "Templates," deals with templates. You have already begun using templates because all the standard containers are template classes. Templates offer a powerful way to reuse code because they enable you to generate functions and classes automatically for each specified type.

Templates

The same code in different contexts can lead to very different operations. For example, `x+y` can mean integer addition, floating-point addition, string concatenation, and a number of other things, depending on the types of `x` and `y`. This is sometimes called *compile-time polymorphism*, to distinguish it from *runtime polymorphism*, which is what happens with classes that have virtual methods. In runtime polymorphism the exact function to call is decided at the last moment, whereas the operation meant by `x+y` is worked out by the compiler.

The C++ template mechanism takes advantage of the fact that operations can be expressed so abstractly. Templates are generators of other types, not types themselves.

In this final chapter, you will learn

- How to write template functions
- How to specialize template functions for a particular type
- How to use parameterized classes
- How to write your own template classes

Generic Functions

Code to do things like search a list, copy values, and so forth will often look exactly the same, no matter what types are being used. The standard algorithms, which you first met in Chapter 3, "Arrays and Algorithms," are examples of what is sometimes called generic programming. Rather than write out the same loops, you can reuse the generic code. In this section you will see exactly how such functions are generated, using function templates.

`sqr()` Revisited

The first example of overloading that we examined in Chapter 6, "Overloading Functions and Operators," involved a pair of functions that were both called `sqr()` but dealt with `int` and `double` arguments. Any type that can be multiplied—not only ordinary types, but mathematical objects such as matrices and complex numbers—can be squared. For each type of object, the `sqr()` function looks exactly the same; thanks to compile-time polymorphism, `x*x` always represents the squaring operation.

The traditional way of efficiently doing this is by using macros. Macros do simple text replacement; for example, `SQR(2.3)` is replaced by `(2.3)*(2.3)` and `SQR(1+2)` is replaced by `(1+2)*(1+2)`. (Note that if you did not include the parentheses, the last expression would be wrong. Appendix C, "The C++ Preprocessor," discusses macros in more detail.) The following code shows how a macro `SQR()` can be defined and used to square both `double` and `int` values:

EXAMPLE

```
;> #define SQR(x) ((x)*(x))
;> SQR(2.3);
(double) 5.29
;> SQR(2);
(int) 4
;> SQR(1+2);
(int) 9
;> int f(int i) {
;1} cout << "called " << i << endl;
;1} return i;
;1} }
;> SQR(f(2));
called 2
called 2
(int) 4
```

However, if the argument to this macro causes a function call or any side effects, these calls or side effects happen twice. Macros are simply too naive for general use. In such cases, a *function template* can be defined, which is parameterized by a type `T`. When the compiler finds the template used as a function, it looks at the type of the arguments. The type parameter `T` is bound to the type of the actual argument (think of the type parameter as a kind of type variable.) Then a new *instance* of the template will be generated, as in the following example:

EXAMPLE

```
;> template <class T>
;>  T sqr(T x)
;>   { return x*x; }
;> sqr(1.2);
(double) 1.44
;> sqr(2);
(int) 4
;> sqr(3.0);
(double) 9.0
;> #v sqr
VAR <void> sqr size 4 offset 9432832
int sqr(int)
double sqr(double)
```

`sqr(1.2)` causes `T` to be bound to `double`, and a function `double sqr(double)` is generated. Likewise, `sqr(2)` causes `T` to be bound to `int`, and `int sqr(int)` is generated. If you then use the UnderC #v command on `sqr`, you will see that two functions called `sqr()` have been generated by the template.

It is useful to think of templates as intelligent macros; the compiler looks at `sqr(1.2)`, notes that the argument is `double`, and can then deduce that `T` must be `double`. It then effectively compiles `double sqr(double x){ return x*x; }`. If you again call `sqr()`, with a `double` argument, the compiler recognizes that this function has already been generated, or *instantiated*.

Function templates are not themselves functions; they are function generators. In engineering, the word *template* refers to a stencil or pattern for cutting metal shapes. Thus, a function template is like a cookie cutter, not the cookie itself. The actual functions generated by the template are called template functions; the order of the words is important.

Function templates can be parameterized by more than one type parameter. Here is `Min()`, which returns the smallest of its two parameters:

EXAMPLE

```
template <class T, class S>
   T Min(T a, S b)
   { return a > b ? b : a; }
;> Min(1,2);
(int) 1
;> Min(2.3,2.0);
(double) 2.
;> Min(2.3,1);
```

```
(double) 1.
;> #v Min
VAR <void> Min size 4 offset 9446992
int Min(int,int)
double Min(double,double)
double Min(double,int)
```

This is a rather simple definition, which has already generated three versions of `Min()`. It seems wasteful; why couldn't you just use one type parameter, `T`? The difficulty is that template argument matching is not too intelligent; if `T` is first bound to `double` by looking at the first argument, the compiler complains if the next argument is an `int`. That is, all arguments declared as `T` must have consistently the same types, and no standard conversions are applied.

A template can be instantiated only if all the operations needed are available. For example, this simple template is intended to save a little typing:

EXAMPLE

```
template <class T>
  void dump(T t) {
    cout << t << endl;
  }
;> int x = 2;
;> dump(x);
2
;> struct S { int a; };
;> S s; s.a = 1;
;> dump(s);
;> CON 4:Could not match void operator<<(ostream,S);
1 parm
CON 4:Could not match void operator<<(void,_Endl_);
0 parm
```

Simple templates like this do not make any assumptions about the type parameters. They trust that they will be called in situations when it makes sense. The plain struct `S` does not overload `operator<<`, so `cout << s` is an error, and the instantiation therefore fails. This also illustrates a general point about templates; they are only fully compiled when needed. Thus, a function template may work fine with some types and give compilation errors with other types.

Programmers also tended to use macros for speed. Small template functions can be inlined and so are just as fast as macros.

Specializations

It is possible to specialize a function template to use different code for certain types. It is rather like function overloading. If the function template takes an argument that still contains a type parameter (for instance, `list<T>`) it is called a *partial* specialization; if it has no type parameters, then it is a *full* specialization. Full specializations of a function template are effectively ordinary functions.

Specialization is very useful when working with the standard containers, which themselves are parametrized types.

The formal arguments of a function template do not have to be simple types. It is useful to use type expressions that involve the type parameters (or "dummy types"). In the following example, there are two function templates called `dump()`. The first would be more efficient than the preceding definition when working with class types; the second `dump()` template handles simple arrays of objects and has an extra size argument:

```
template <class T>
  void dump(const T& t) {
     cout << t << endl;
  }

template <class T>
  void dump(T t[], int n) {
    for(int i = 0; i < n; i++) cout << t[i] << ' ';
    cout << endl;
  }
```

These two templates can easily be distinguished from each other because they have different numbers of arguments, so a kind of overloading is possible. In the case of templates, this is called *specialization*.

If you now call `dump()` with a `std::string` argument, C++ will use the first template and can deduce that `T` is `std::string` but will then pass that argument as `const std::string&`. Likewise, `dump(arr,4)` (where `arr` is an array of integers) matches the second template and leads to `T` becoming `int`.

Note that the arguments of a function template can sometimes be ordinary types, which are converted in the usual fashion; the second version of `dump()` has an argument `int n`. However, each dummy parameter (in this case, `T`) must be mentioned at least once. If not, C++ cannot deduce what `T` must be from the arguments alone.

Working with the Standard Containers

As you may have guessed, the standard containers such as `std::list` are parameterized types that are defined as templates. A powerful use of function templates is to generate functions that can operate on any list. Here is a function template for dumping a list:

EXAMPLE

```
template <class T>
  void dump(const list<T>& ls) {
    list<T>::const_iterator li;
    for(li = ls.begin(); li != ls.end(); ++li)
       cout << *li << ' ';
    cout << endl;
  }
;> list<int> ls;
;> ls.push_back(1); ls.push_back(2);
;> dump(ls);
1 2
```

Please note that the list iterator type used here is `list<T>::const_iterator`. The difference is that a `const_iterator` cannot be used to modify the list. Since the list argument is `const`, C++ will not allow you to use a plain iterator here, because it cannot guarantee that it won't be used to change any list values. This can be tricky, because it's easy to forget this requirement (I did it for the first version of this template!). The error messages from C++ compilers can sometimes be a little obscure. If you used iterator instead of const_iterator earlier, BCC32 would give the following error message:

```
Error E2034 memtemp.cpp 36:
 Cannot convert 'list<int,allocator<int> >::const_iterator' to
 'list<int,allocator<int> >::iterator'
 in function dump<int>(const list<int,allocator<int> > &)
```

The first thing is to concentrate on the first error (subsequent errors are often for the same reason or simply because the compiler is confused). Mentally remove the `allocator<int>` and you get `list<int>`; this is a default argument that you may never need to change. However, it does make reading error messages a bit tiresome. So the compiler is telling us that it cannot convert `list<int>`'s `const_iterator` to `iterator`; this error refers to the assignment `li = ls.begin()`. Since `ls` is a `const` reference, `ls.begin()` is a `const` type; hence the problem. The GCC error message got from running the C++ command is rather strange:

```
memtemp.cpp: In function `void dump<int>(const list<int,allocator<int> > &)':
memtemp.cpp:49:   instantiated from here
memtemp.cpp:36: no match for `_List_iterator<int,int &,int *>& =
                 _List_iterator<int,const int &,const int *>'
```

GCC first tells us where the function template was originally called, and then gives the actual error inside the template. The problem is with an assignment; and the clue is that on the right-hand side the type parameters are `const`. Not obvious, but you start to recognize a pattern.

I don't want to scare you with nasty C++ error messages, but you are going to meet them in real life, and you have to learn to interpret what the compiler is trying to say. Error messages to do with templates and especially with the standard library can be hairy. The main thing is not to be intimidated. This is one good reason, incidently, why the standard algorithms are often easier than loops. Here is another version of the same template, this time using `for_each`:

```
template <class T>
  void dump_item(const T& v) { cout << v << endl; }

template <class T>
  void dump(const list<T>& ls) {
     for_each(ls.begin(),ls.end(),dump_item);
  }
```

It is perfectly fine to pass function templates like `dump_item()` to standard algorithms like `for_each()`, although I could not use `dump()` itself because it would not be clear which template was meant.

The convenience of the `dump()` template is that it can operate on any list, provided that its element type can be written out (that is, `operator<<` has been overloaded for that type.) This works well for `list<string>`, `list<double>`, and so on.

It also coexists happily with the simple `dump()` templates defined in the preceding section. Consider the call `dump(ls)` where `ls` is some `list` type. The system finds three templates called `dump()`, two of which can match `list<int>` because they each take one argument. The third template matches `list<int>` even more closely, so it is chosen; this template is said to be more specialized than the others. That is, although `list<int>` can match the `const T&` pattern, the `const list<T>&` pattern is a closer fit. By comparing the actual type parameter (`list<int>`) with the formal type parameter (`const list<T>&`), the compiler can deduce that `T` must be `int`.

In general, specialized templates give you less trouble than general templates. All the assumptions required by the dump() template specialized for `list` (the nested type `iterator`, the methods `begin()` and `end()`) are guaranteed by the fact that the function argument must be a `list` type.

Functions That Operate on Any Sequence

A major use of function templates is with parametrized classes like `list`. As you saw in the last section, specialized templates can be defined that will work with a container class such as `list`, for any type. The standard algorithms themselves are all function templates, so that they can be used with arguments of any type.

The idea of input and output sequences is very important in understanding and using the standard algorithms.

Sequences and `for_each()`

The standard algorithm `for_each()` is very simple. In the following implementation, it is given an input sequence and a function to apply to each element of that sequence:

EXAMPLE

```
template <class In, class Fn>
  void for_each(In i, In iend, Fn f) {
     for(; i != iend; ++i) f(*i);
  }
;> void show(int i) { cout << i << endl; }
;> for_each(ls.begin(), ls.end(), show);
1
2
;> for_each(ls.begin(), ls.end(), dump);
1
2
;> list<int>::iterator li;
;> for(li = ls.begin(); li != ls.end(); ++li) dump(*li);
1
2
```

NOTE

You can pass template functions (such as `dump()`) as well as conventional functions to standard algorithms such as `for_each()`.

This example shows three ways of saying the same thing, and the last way involves explicitly writing out the loop. It is clumsy because you must explicitly declare a `list` iterator. In contrast, when you instantiate `for_ each()`, the compiler binds the type parameter `In` to the type `ls.begin()`, which is in fact `list<int>::iterator`. `for_each()` simply assumes that it is passed objects that behave like forward iterators. That is, `*i` gives the current value, and `++i` moves to the next value. These are precisely the properties of C++ pointers.

NOTE

With modern compilers, the `for_each()` version is just as fast as the explicit loop. In fact, it may have a slight edge because `ls.end()` is evaluated only once.

Standard Algorithms

The idea behind standard algorithms is to avoid writing out loops, which are "tedious and error prone," as Bjarne Stroustrup says in *The C++*

Programming Language. However, typing `ls.begin(),ls.end()` all the time seems equally tedious. Why is `for_each()` not defined as follows?

```
template <class C, class Fn>
  void for_each(const C& c, Fn f) {
    typename C::iterator i = c.begin(), iend = c.end();
    for(; i != iend; ++i) f(*i);
  }
;> for_each(ls,dump);
1
2
```

NOTE

When the compiler is doing an initial pass through the template code, it doesn't know that `C::iterator` is a valid type, so you have to give it a hint. That is why you use `typename`.

This example indeed does the job for any standard container or for any type that shares this part of the container interface (such as `std::string`). But these assumptions are too strict. The following example shows two things that the standard `for_each()` can do that the preceding version cannot. First, the standard `for_each()` happily operates on built-in arrays. Second, it's very useful to specify the input sequence. The second `for_each()` applies its function to every element after the first occurance of 10:

```
;> int arr[] = {1,10,3};
;> for_each(arr,arr+3,dump);
1
10
3
;> int *p = find(arr,arr+3,10), p_end = arr+3;
;> for_each(p,p_end,dump);
10
3
```

The standard algorithms efficiently use compile-time polymorphism. All standard iterators behave like C++ pointers, so ordinary pointers work fine as well. The standard library emphasis is on both speed and elegance. But an old engineering proverb says you can only have two out of three desirables. Aggressive optimization for speed, involving wholesale inlining, can produce bulky code. One solution to this problem is outlined in the section "The Standard Containers as Class Templates."

Objects That Act Like Functions

C programmers are pleasantly surprised to discover that `std::sort()` can be considerably faster than the old-fashioned `qsort()` routine. The reason is, again, that the compiler can often inline a function directly into the sort algorithm. In fact, any standard algorithm that requires a function is satisfied by anything that behaves like a function. In C++, the action of calling a function is an operator, and it can be overloaded. That is, if `obj` is some object of class `C`, `obj()` can be defined to mean anything we like by overloading `C::operator[]`. The following is a class that behaves like a function:

EXAMPLE

```
class Sum {
  double m_sum;
public:
  Sum()
   : m_sum(0.0) {}

  void   operator() (double v) { m_sum += v; }

  double value()
  { return m_sum; }
};
;> Sum s;
;> s(2.0);
;> s(3.0);
;> s.value();
(double) 5.0
;> double arr[] = {2.0,3.0};
;> Sum t;
;> for_each(arr,arr+2, t);
;> t.value();
(double) 5.0
;> #include <numeric>
;> double val = 0.0;
;> accumulate(arr,arr+2,val);
(double) 5.0
```

Templates turn this party trick into a genuinely useful technique. Remember that templates are instantiated through a sophisticated kind of macro substitution that binds the function parameter `Fn` to the type `Sum`. The expression `f(*i)` is finally compiled as the method call `f.operator()(*i)`. The advantage of function-like objects is that they carry their own context around with them. With ordinary functions, finding the sum would require a global variable.

In the example, a specific numerical algorithm (`accumulate()`) does the same job as using `for_each()` with `Sum`. You need to pass it a final argument so that the compiler can deduce what the return type must be. Generally, you should use the most specific algorithm you can find.

Class Templates

Function templates allow very general code to be written that will work for different kinds of types. Class templates do the same for classes; all the class data and member functions become parameterized by type parameters. Class templates are machines for generating families of related classes.

A Parameterized Class

When you are designing classes, you need to make choices. In some cases, such as when either choice seems perfectly valid, it is irritating to have to choose. For example, consider a `Point` class that represents a two-dimensional coordinate. Should it contain integers, floats, or doubles? There are good arguments for all three choices. Using `typedefs` is helpful, as in the following example:

```
typedef double coord_t;
struct Point {
   coord_t x,y;
};
```

In this example, there is only one place where the explicit type is mentioned (this is particularly useful for numerical code, where the choice of `float` or `double` can give you valuable clues about the precision of the algorithms). However, sometimes you want floating-point coordinates, and other times you want integer coordinates. Once again, macro magic such as the following used to be applied:

```
#define POINT(T) Point_#T
#define DEF_POINT(T) struct POINT(T) { T x,y; }
DEF_POINT(int);    // => struct Point_int { int x,y; };
DEF_POINT(float);  // => struct Point_float { float x,y; };
POINT(int) p1,p2;  // => Point_int p1,p2;
```

This magic is fine, to a point, but it scales badly. Imagine taking the `Vector` class mentioned in Chapter 9, “Copying, Initialization, and Assignment,” and making a huge 200-line `#define` statement. If there were an error, you would have virtually no idea where it occurred, and you wouldn’t be able to debug any runtime problems. The template solution is much more elegant:

EXAMPLE

```
template <class T>
  struct Point {
    T x,y;
  };
;> Point<int> ip;
;> Point<double> dp;
```

NOTE

The two terms *class template* and *template class* are sometimes confused, but they are different beasts. *Class template* is like the cookie cutter and *template class* is like the cookie. A class template is a machine for producing template classes, such as `Point<int>`, which behave in absolutely the same way as ordinary classes.

It is important to note that `Point<int>` and `Point<double>` are also completely independent classes. It is true that they are both types of `Point` and that `int` can convert to `double`. But these facts don't mean that a `Point<int>` can automatically convert into a `Point<double>`. You can easily write functions to do the conversion, as in the following example:

```
Point<double> point(Point<int> p) {
   Point<double> pd;
   pd.x = p.x;  pd.y = p.y;
   return pd;
}

Point<int> point(Point<double> pd) {
   Point<int> p;
   p.x = pd.x;  p.y = pd.y;
   return p;
}

;> Point<int> p; p.x = 10; p.y = 20;
(int) 10
(int) 20
;> Point<double> pd = point(p);
;> p.x; p.y;
(double) 10
(double) 10
```

Observe that the `Point<int>` and `Point<double>` functions are quite distinct from one another because their argument types are different classes. Later in this chapter, the case study shows how you can build such relationships between template classes.

The template type parameters can be ordinary types as well. Consider the following `Buffer` class, which has two type parameters; the first can be any C++ type, and the second has to be an `int`:

EXAMPLE

```
// buffer.cpp
// C++ By Example, Chapter 10, Steve Donovan
#include <iostream>
using namespace std;

template <class T, int N>
```

```
  class Buffer {
   T m_arr[N];
  public:
   Buffer()
   { for(int i = 0; i < N; i++) m_arr[i] = 0; }

   int size()
   { return N; }

   T& operator[] (int i)
   { return m_arr[i]; }
 };

template <class T, int N>
 ostream& operator<< (ostream& os, Buffer<T,N>& p)
 {
  for(int i = 0; i < p.size(); i++) cout << p[i] << ' ';
  return os;
 }

int main()
{
 int vals[] = {23,43,56,22,22};
 Buffer<int,10> buff;
 for(int i = 0; i < 5; i++) buff[i] = vals[i];
 cout << buff << endl;
}
```

The `Buffer` class template is meant to be an efficient array-like class. The parameter `N` is a constant, so a C++ array can be used as the representation. Note how `Buffer` template classes are written: `Buffer<int,10>`. The template class `Buffer<int,12>` would be completely different class.

It is often not difficult to convert existing classes into template classes. Generally, it's a good idea to develop code for some particular type, iron out the troubles, and only then make the code into a template. You can turn the `Vector` class from Chapter 9 into a class template by adding a few extra lines. Here is the first forty-odd lines of a `Vector` class template; the rest of the code is absolutely identical, and you can see the whole class in `chap10\vector.h`. There are three changes from the original version: `template <class T>` has been added before the class declaration (see (a)), `typedef int T` is commented out (see (b)), and the exception type `RangeError` has been moved out of the class body. This conversion to a template was easy because the original `Vector` uses `typedef` to define `T` to mean `int`.

```
// C++ By Example, Chapter 10
// A simple vector template
#include "ref_count.h"
const int NGROW = 10;

struct RangeError { // *moved outside
     int idx;
     RangeError(int i) { idx = i; }
     int where() { return idx; }
};

template class<T>   // *added (a)
class Vector: public RefCount {
 public:
//  typedef int   T;   // *removed (b)
 private:
   T *m_begin;
   T *m_end;
   int m_size;
   int m_cap;
 public:
   typedef T *       iterator;
   typedef const T * const_iterator;
   typedef T         value_type;

   iterator begin()  { return m_begin; }
   iterator end()    { return m_end;   }
   int size()        { return m_size; }
   int capacity()    { return m_cap;  }

   T& operator[] (int i)
   { return m_begin[i]; }

   T& at(int i)
   {
    if (i < 0 || i >= m_size) throw RangeError(i);
    return m_begin[i];
   }
  . . . (rest unchanged) . . .
```

A template class can inherit from an ordinary class. This can be very useful from both design and implementation perspectives. From the design perspective, it means that all the classes `Vector<T>`, (for all `T`) are related to each other by derivation; they are all `RefCount` objects. The class template `Vector` becomes a machine for breeding subclasses of `RefCount`. In implementation terms, the benefit is that you do not have to carry the extra reference-counting code in each instance of Vector<T>. With large classes, factoring out the common, non-type-specific code and moving it up into a base class can make a noticeable difference.

Moving out RangeError is necessary because you want one exception class to represent out-of-range errors for all instances of Vector. If RangeError were left inside, Vector<int> would throw Vector<int>::RangeError and Vector<double> would throw Vector<double>::RangeError; these are quite distinct types (unless you needed to make the exceptions distinct). So, in fact, nested classes are implicitly template classes.

The rest of the code for Vector<T> is completely the same as for Vector, but the meaning is different. For example, look at the copy assignment operator for Vector<T>:

```
Vector& operator= (const Vector& v)
{
     _alloc_copy(v);
     return *this;
}
```

Within the class body is the one place you can get away with just using the class template name to mean the template class; in this case, it is assumed that Vector means Vector<T>. However, it would not be an error to spell it out in full.

The Standard Containers as Class Templates

It is clear from the Vector<T> example in the preceding section how std::vector<T> can be implemented. But before we move on, we need to review how things were handled before templates were available. The first problem with creating containers without templates is that they are not type safe. Imagine if you had only list<void *>; any pointer could be put in such a list. You would need to write code like this:

```
typedef list<void *> List;  // the one & only list…

void draw_shapes(TG& tg, const List& ls) {
   List::iterator li;
   for(li = ls.begin(); li != ls.end(); ++li)
      static_cast<Shape *>(*li)->draw(tg);
}
```

This looks nasty, and it is nasty. There is no guarantee at runtime that this list contains only pointers to Shape. A program would inevitably decide to put something odd in the list when it was being used for some crucial operation on the other side of the planet (or, indeed, another planet altogether). It is not possible to check all the possibilities in even a moderate-sized application. Type safety is a guarantee that no surprises with odd types can happen—that they will be caught by the compiler, not the customer.

Traditional object-oriented languages rely on runtime type information. Every class derives from some polymorphic base class, usually called `Object`. In Delphi, for instance, deriving from `Object` is implicit, so that the `is` operator can always work. Of course, this strategy works in C++ as well; the ultimate base class `Object` needs at least one virtual method. The class `Shape` will have to be derived from `Object` in some way. Then `dynamic_cast()` will work. The previous example becomes this:

```
typedef list<Object *> List;  // the one & only list…

void draw_shapes(TG& tg, const List& ls) {
   List::iterator li;
   for(li = ls.begin(); li != ls.end(); ++li) {
    Shape *ps = dynamic_cast<Shape *>(*li)
    if (ps != NULL) ps->draw(tg);
   }
}
```

There is now a proper runtime guarantee, at the cost of continuous checking. But what should you do if the object isn't a `Shape` pointer? Surely you should raise an alarm or make a note somewhere. Although this code is safe, it could be masking an error somewhere else. There should only be `Shape` pointers in this list; it isn't considered particularly clever to keep different kinds of objects together.

So far, we have only talked about pointers. Languages such as Java and Delphi really deal only with references to objects, but they also have to deal with the common case of just wanting a list of integers or doubles. You can either get a whole number of extra container classes or wrap the simple types themselves as objects, which is what Java does (Delphi programmers have a horrible habit of trying to stuff integers into lists of pointers). This is the second problem related to life without templates.

C++ must deal with value-oriented types such as `std::string`. These types do not fit very well into a pointer-list scheme. There is an object-oriented approach to containers of such types, but it isn't pretty. You define a special base class—which you could call `Linkable`—that contains a pointer to another `Linkable` object. Any class that you inherit from `Linkable`, therefore, has the ability to link to another `Linkable` object, rather like the parade of circus elephants holding the tails of their fellows. You can then run through the list by following pointers, until some stopping condition (in the case of a circular list, the elephants keep going around the ring). But this is awkward;

it means that you have to decide, as part of your design, that your objects are going to be kept in a list. There would need to be auxiliary types such as linkable strings.

The design of the C++ standard containers answers these two problems, as well as another: C++ standard containers present a very similar interface to the world. With some care, code can switch from using `list<T>` to `vector<T>` by changing a single `typedef`. Say you have found that a time-consuming task is iterating through containers; in this case, switching to a `vector` makes perfect sense. The philosophy is that whether you put widgets in a `list`, `vector`, `deque`, `map`, or `set`, it should not be a high-level design decision.

Note an important property of class templates; not every method is compiled when a template class is instantiated; only methods that are referenced in the code, or are needed to implement those methods, are compiled. This can obviously save a lot of dead code (although smart linkers can strip out such baggage). But this is crucial to the design of classes like `list<T>`. For example, the standard list has a method `remove()` that is given a specific element value to take out, but not every type has defined what it means to be equal. Similarly, the method `sort()` assumes that `e1 < e2` makes sense. So instantiating only code that is directly needed makes it possible to generate lists of simple structs, which don't define equality or ordering.

The downside of using template containers is that a program might end up containing many different instances of a particular template type. It is common to keep lists of many different kinds of objects, and it is unfortunate if the program has to keep that many (practically identical) copies of the list code. This is often solved by specialization: You can define `list<T *>` in terms of `list<void *>`. The specialized template for pointer types becomes a thin "type-safe" wrapper around `list<void *>`. The code would look something like this:

```
typedef list<void *> ListP;
template <class T>
 class list<T *>: private ListP {
  ...
  void push_back(T *p)
  { ListP::push_back((void *)p); }

  T  * back()
  { return (T *) ListP::back();  }
 ...
 };
```

The basic pointer list type `ListP` is used as a private base class because `list<T *>` redefines every public member anyway—it is pure "implementation inheritance." This is not the most exciting code to write, but such specializations are fast because they are so easy to inline.

Template Functions That Generate Template Classes

One big difference between class and function templates is that the type parameter is not deduced from the arguments for class templates. That is, you always have to follow the template name with `<type>` when constructing a template class. At first this seems irritating, compared to template functions that work so transparently. The reason is contained in the question: What arguments can be used to deduce the type? You would have to use the constructor arguments, but there will always be more than one constructor in a non-trivial class and so practically it's impossible to deduce type parameters for a class template. However, it's not difficult to write function templates that deduce the type and use it to instantiate a template class.

For instance, an interesting part of the standard C++ library is `back_inserter()`, which makes copying into containers easy. Here is an example:

```
;> list<int> ls;
;> int arr[] = {20,30,50};
;> copy(arr,arr+3,back_inserter(ls));
```

`back_inserter()` is necessary because originally the list has no elements; using `ls.begin()` as the output iterator will not work, since there simply aren't any elements in the list. Something is needed that looks like an output iterator and does the pushing at the list's end.

To do this, you create a class such as the following that imitates an output iterator—that is, it defines `operator*` and `operator++`. This class keeps one member, which is a reference to the container `m_container`, and another member, which is the same type as the container element `m_val`. `operator*`. This class also returns a reference to `m_val`, so it will get modified by expressions like `*pi = val`. `operator++` and then push `m_val` onto the end of `m_container`:

EXAMPLE

```
template <class C>
 class BackInserter {
 private:
    C& m_container;
    typename C::value_type m_val;
  public:
   typedef C::value_type value_type;
   BackInserter (C& c)
    : m_container
    { }

   value_type&
   operator*()
   { return m_val; }

  BackInserter&
   operator++()  {
     m_container.push_back(m_val);
     return *this;
   }
 };
;> BackInserter<list<int> > bi(ls);   // watch out for '> >' !
;> *bi = 10;      // sets the value...
(int&) 10
;> ++bi;          // actually does the push!
```

This example probably isn't the fastest way to handle the situation, but it does the job. This class depends on two reliable things: A standard container makes its element type known as `value_type`, and it has a `push_back()` method. But this kind of class is not intended for direct use. Declaring it is a bit painful because it is itself parameterized by a template class. (Notice the extra space between the last angle brackets (> >)—this is one of the few places in C++ where a little whitespace is necessary; the symbol >> is something else altogether). Therefore, `back_inserter()` is defined as a simple function template:

```
template <class C>
   BackInserter<C>
   back_inserter(C& c)
   {
      return BackInserter<C>;
   }
```

This is a common technique for generating template classes. The function template `back_inserter()` deduces the type parameter and uses it to construct the `BackInserter` object. Again, using `back_inserter()` is going to be just as fast as writing the loop out fully.

Separating the Template Interface from the Implementation

As with regular classes, with class templates, it is possible to separate the interface from the implementation. Here is a simple `Point` template class, separated into interface and implementation:

```
// point.h
template <class X>
  struct Point {
   X x,y;

   Point(int _x, int _y);
   void set(int _x, int _y);
  };

// point.cpp
template <class X>
  Point<X>:: Point(int _x, int _y)
   { set(_x,_y); }

template <class X>
  void Point<X>::set(int _x, int _y)
  { x = _x; y = _y; }
```

The member function definitions are templates, as you would expect, except that the class type (`Point<X>`) contains the dummy type. As with ordinary classes, the class body must have previously declared the method. However, currently most compilers have difficulty with the separate compilation of templates. So in practice, the definitions of the methods must be included directly as well.

Member Templates

Member templates are an interesting and occaisionally very useful feature of modern C++. Member functions can themselves be defined as function templates. This example is an updated version of the `Buffer` class template defined earlier. Note that the `assign()` method is itself a template:

EXAMPLE

```
...
template <class T, int N>
  class Buffer {
   T m_arr[N];
  public:
   template <class In>
     void assign(In start, In finish) {
       int i = 0;
       for(; start != finish; ++start) m_arr[i++] = *start;
     }
   Buffer()
   { for(int i = 0; i < N; i++) m_arr[i] = 0; }

   int size()
   { return N; }
   T& operator[] (int i)
   { return m_arr[i]; }
 };
...
int main()
{
 int vals[] = {23,43,56,22,22};
 Buffer<int,10> buff;
 buff.assign(vals,vals+5);
 cout << buff << endl;
// standard list<int> defines assign() as a member template!
list<int> ls;
 ls.assign(vals,vals+5);
 dump(ls);
}
```

OUTPUT

```
23 43 56 22 22 0 0 0 0 0
23 43 56 22 22
```

The `assign()` method in fact looks rather like the standard algorithm `copy()`. It is also parameterized by some input iterator type; in the example it's used to copy an array of integers into the buffer, so the iterator type is `int *`. But you could use `assign()` with any valid input sequence.

The standard containers all have a method `assign()`, which is defined as a member template.

NOTE

Historically, templates have been the most difficult part of C++ to get right. Currently, UnderC does not support separate member function definitions or member templates, but it will in the future.

Case Study: Smart Pointers

Chapter 9 mentions some problems with using dynamically allocated objects in C++. Eventually, someone must remember to dispose of them. If, however, they are thrown out before someone is finished with them, there is trouble. (People with small children should recognize the general problem.) One common solution is reference counting, which you saw working behind the scenes of the `Array` class in Chapter 9. It's clear when a particular object still has users, and premature disposal cannot happen.

However, to use reference-counting you have to do a certain amount of work. Whenever an object keeps a pointer to a reference-counted object, it must increment the reference count. It is equally important to call `dispose()` afterward. It would be nice if this could happen automatically. This can be done, and the extra cost is not too great. You need to define *smart pointers*, which are objects that behave like object pointers and can be very useful. The standard library provides such a pointer, called auto_ptr<T>, and here is a simplified implementation of it:

```
template <class T>
  class auto_ptr {
    T *m_ptr;
  public:
    auto_ptr(T *p) : m_ptr(p) {}
    ~auto_ptr() { delete m_ptr; }

    T& operator* () const { return *m_ptr; }
    T* operator->() const { return m_ptr; }
    operator T * () const { return m_ptr;  }
 };

void test_ptr() {
  auto_ptr<Person> pp (new Person());
  pp->set_name("Billy");    // operator->
  read(pp);                 // can match Person * (conversion)
  if (pp->illegal()) throw Person::Bad();
  dump(*pp);                // operator* —- Person&
}  // pp is destroyed.
```

In this example, the class auto_ptr is stripped down to its basics; it is a wrapper around an object pointer m_ptr. When pp (which is of type auto_ptr<Person>) is automatically destroyed at the end of test_ptr(), its destructor disposes of m_ptr. Otherwise, it behaves just like a Person * object because of the operator overloads. No matter how you leave the function, test_ptr(), pp is destroyed, and its pointer is deleted. (The full version of auto_ptr<> also manages who gets to delete the pointer, but that's not relevant here.)

The overloads are simple, but operator-> is one we haven't talked about yet. Normally, an object is not followed by ->, but if it is, the compiler looks for an overloaded operator->. If the compiler finds such an operator, it looks at the return type, and if the return type is T *, the compiler looks for any members of T after ->.

By using this technique you can fashion pointers with any behavior you choose. It is simple to track access to objects, for example. The idea is to design a smart pointer that makes using reference-counted objects easy, so all objects in the system must derive from RefCount. Here is a smart pointer ptr that manages an object pointer derived from RefCount. The method set() is the interesting one:

```
// ptr.h
template <class T>
 class ptr {
   T *m_p;
 public:
   void set(T *p) {
     if (m_p != NULL) m_p->dispose();
     m_p = p;
     if (m_p != NULL) m_p->incr_ref();
   }

   ptr(T *p = NULL)
    : m_p(NULL)
    { set(p);    }

   ptr(const ptr& pp)
   : m_p(NULL)
   { set(pp.m_p);  }

   ptr& operator= (const ptr& pp) {
     set(pp.m_p);
     return *this;
   }
   ptr& operator= (T *p) {
     set(p);
     return *this;
   }

   void unique()
   { set(new T(*m_p)); }

   bool is_empty() const { return m_p == NULL; }

   T& operator*  () const { return *m_p; }
   T* operator-> () const { return m_p; }
   operator T *  () const { return m_p; }

  };
```

The last three overloads in this class give you the smart pointer interface. The business of managing reference counting is managed by the `set()` method, which works as follows. If the `ptr` object already has a pointer, you call `dispose()`. (Remember that this does not necessarily delete the object; it does so only if it has no other users.) The new pointer's reference count is then incremented, and the smart pointer is ready for business.

`unique()` allows you to force a unique copy of the pointer. It depends on a sensible copy constructor being available, that is, that the constructor `T(const T&)` does generate a proper copy of the object. Here is some example code that shows how `ptr` is used. Note that you may freely pass `ptr<Person>` objects around; they behave just like `Person*`, but quietly dispose of `Person` pointers when they are no longer needed:

```
void test1(ptr<Person> p) {
  ptr<Person> p_new(new Person(p));
  read(p_new);
  dump(p_new);
};  // p_new's object is deleted
    // p's isn't (still owned)

ptr<Person> test2() {
  return ptr<Person> (new Person());
}

void test2() {
 ptr<Person> p1(new Person());
 test1(p1);
 ptr<Person> p2 = test2();
} // p1 & p2 destroyed.
```

This example shows that `ptr<Person>` manages the life of the `Person` pointer. If this pointer is consistently used, there is no need for an explicit `delete p`. Deleting from a list of such objects would also cause disposal, as with value-oriented types such as `std::string`. But `ptr<Person>` still obeys pointer semantics; that is, assigning a `ptr<Person>` to another `ptr<Person>` merely shares a reference to the same `Person` pointer, without any potentially expensive copying taking place. This case study shows that it is possible to use pointers in a safe fashion in C++.

Do these smart pointers maintain the same relationship between their pointers? That is, does a `ptr<Employee>` match a `ptr<Person>`, given that `Employee` is a derived class of `Person`? This code demonstrates that it is still possible to assign `ptr<Employee>` to `ptr<Person>`:

```
;> ptr<Employee> pe (new Employee());
;> ptr<Person> pp;
;> pp = pe;            // cool!
;> pe = pp;            // bad!
```

The correct assignment in this example takes place in two steps. Obviously, no assignment operators can do this directly, because `ptr<Employee>` is not related to `ptr<Person>`. When no standard conversion is possible, C++ will look for user-defined conversion operators. `ptr<Employee>` has a user-defined conversion operator `operator Employee*`, and so `pe` will be converted to an `Employee` pointer.

The next step happens because an `Employee` pointer can match a `Person` pointer, so the compiler can use `Person`'s assignment operator `operator=(Person *)`. Maintaining the relationship between the various pointer types is the crucial part of using `ptr`.

What's Next

Templates are powerful tools, and there's much more to them than what I've discussed. This book is meant to be an introduction to the world of C++, not the definitive guide book. My favourite C++ tutorial book has been Stanley Lippman's *C++ Primer*, 3rd edition, and of course the best reference is from the C++ man himself, Bjarne Stroustrup. *The C++ Programming Language* is an essential part of any serious programmer's bookshelf. (Both of these books are from Addison-Wesley.)

C++ is a very rich language with many ways of saying the same thing. Some critics believe this is a bad thing, since programmers become confused about which approach to use. However, this is also true about natural languages like English (which has far too many words meaning almost the same thing), and the solution is to learn good C++ style. Style depends on context; if I am experimenting with code interactively with UnderC, I will use macro shortcuts. They are also appropriate with "quick and dirty" code you write for your own purposes (like specialized tools). But they are not appropriate for programs which are worked on by other people; simularly, another rule is not to use `using namespace xxx` (where `xxx` is some namespace, usually `std`) in header files. It is useful to remember that a program is a public document that is meant to be read by humans as well as computers.

The best way to learn good style is to read lots of good C++ code and make lots of mistakes. This sounds like a joke, but it isn't: learning from your mistakes is the main learning skill you will need. Mistakes are an opportunity to learn something new. Also, learn from the C++ community; there are many C++ resources available on the Net. I have provided links to some of these pages on the C++ By Example site: see `http://home.mweb.co.za/sd/sdonovan/ccbx.htm`.

Part III

Appendices

A

UnderC for Windows (UCW) Command Reference

Loading and Running Programs

The UCW (UnderC for Windows) console window allows you to enter C++ statements and UCW commands at the ;> prompt. Generally, expressions and statements are terminated with a semicolon; anything beginning with #, like preprocessor directives, and UCW commands are not. Warning and informational messages (including values of variables) in the console appear in green, and error messages appear in red.

Using the Clipboad with UCW

By dragging the mouse over the console to highlight it, you can mark an area for copying to the Windows Clipboard. After an area is highlighted, you can use Ctrl+C to copy the content as you would in other Windows applications. If you want to save the whole session to a file, you can use the log command (`#log`). It is also possible to execute commands and statements by pasting them into the console with Ctrl+V.

Exiting UCW: `#q`

Normally, you use `#q` to quit a session, but there is also `#ql`, which means "quit and write to the log file." The resulting automatic log files are made up of the date and time, like this: 0805-1645 (which represents month, day, hour, and minute).

Changing and Displaying the Working Directory: `#cd`, `#pwd`

It is useful to know where you are. Any program has a *current working directory*, which determines the default place where files will be found. UnderC has a command `#pwd` that will tell you what the working directory is, and a command `#cd` that works like the DOS or UNIX command of the same name:

```
;> #pwd
C:\gcc\test
;> #cd c:\bolsh\ucw\examples
;> #pwd
c:\bolsh\ucw\examples
;> #cd /gcc/test
;> #pwd
c:\gcc\test
;> #cd ../bin
;> #pwd
c:\gcc\bin
```

In Windows programs, note that you can actually use the old UNIX slash (`/`) instead of the backslash when writing paths to files. The backslash (`\`) is only really necessary at the DOS or NT command prompt. You should use / in your paths if you want your programs to be portable.

Loading Code and Running Programs: #x, #l, #r

You can execute system commands by using the execute command (#x). For example, `#x dir /w *.cpp` lists all C++ source files.

You must load code before you can run it, and this is the job of the #l command, which is usually followed by the source file. The source file can have any extension you like, but most source files end in .cpp or .h. If you have files that contain UnderC extensions, then the convention is to use .uc, to distinguish the file from files that are compatible with the standard C++ language. UnderC stops when it hits the first error because it concentrates on sorting out one error at a time. (Errors in C++ files often cause a cascade of unrelated errors after the first fault.) Warnings are not fatal, but they usually try to tell you something. Error messages will tell you the file and the line number where the error was encountered, as in this example:

```
;> #l inc1.cpp
inc1.cpp 32:parse error
inc1.cpp 32:Cannot find 'w'
```

If the #l command is not followed by a file, UCW assumes that you are reloading the last file.

When code is loaded and compiled, it can be run in two ways. First, by typing an expression that involves a function, you can exercise each function in the file. Second, if the file contains a `main()` function, you can run the program by using #r, followed by the command-line arguments, as you would type them at a Windows command prompt.

Of course, you can just type `main();` at the ;> prompt, but using #r has several advantages: First, the program input and output both occur in a separate console window, which you can interrupt by using the stop command (#s) if you get into an infinite loop. Second, you can also inspect program variables from the ;> prompt if they are not local to some function. Therefore, it is useful keeping some variables global while you are debugging a program.

There are no restrictions on what you can do when a program is running. For instance, say your program consists of the file one.cpp, which exports `do_it()`, and main.cpp, which calls `do_it()` by using values that are read in from standard input. After loading both of the files, you can run the program and exercise `do_it()`; at any point, you can modify and reload one.cpp without having to stop the program. A rule of thumb is that you should not modify the function you are currently running. You can get away with modifying main.cpp in this case, but the program will probably crash (albeit rather harmlessly) when `main()` gets to the end. You can modify any global variables at any point.

Inspecting and Browsing symbols: #v, #d, #lv

Several UCW commands are useful in a number of situations. The module command (`#mod`) has two forms (a module is another name for a loaded C++ source file.) If you type `#mod` without parameters, you get a list of all files currently loaded. If you give `#mod` the name of a function, it will tell you in which module that function is defined and at which line.

The variable inspect command (`#v`) gives you information about a symbol, if it exists. It tells you the type (and size) of a variable, as well as the prototypes available for a function name—(there can be a number of overloaded versions of the function). You can use the `scope` operator (`::`) to qualify the reference to a method or a function within a namespace. The following are the usual prototypes available for the overloaded `operator<<`, and the definition of `string::npos`:

```
;> #v operator<<
VAR ostream& operator(ostream&,int) << size 4 offset 9303040
ostream& operator<<(ostream&,int)
ostream& operator<<(ostream&,double)
ostream& operator<<(ostream&,float)
ostream& operator<<(ostream&,const char*)
ostream& operator<<(ostream&,_Endl_&)
ostream& operator<<(ostream&,void*)
ostream& operator<<(ostream&,char)
ostream& operator<<(ostream&,const string&)
;> #v string::npos
VAR const int string::npos size 4 offset 3012
2147483647 was value
```

The display structure command (`#d`) is used to inspect the members of a structure, and the list variables command (`#lv`) will show the currently defined variables in scope:

```
;> struct Point { int x,y; };
;> Point p;
;> p.x = 10; p.y = 20;
(int) 10
(int) 20
;> #d p
(int) x = 10
(int) y = 20
;> int x=1,y=2,z=3;
;> #lv
(int) _xo_ = 0
(Point) p = Point {}
```

```
(int) x = 1
(int) y = 2
(int) z = 3
;>
```

Setting Breakpoints

Debugging programs is unavoidable and is much easier if you can make execution stop at any given line in a program, to examine (that is, to *inspect*) the values of variables. You can do this by setting a *breakpoint*. For example, say you want to know why the following program (`main.cpp`) isn't doing what it's supposed to do (note that the line numbers are just for reference):

```
1 #include<iostream>
2 #include<cstdlib>
3 using namespace std;
4 int main(int argc, char *argv[]) {
5   int count = atoi(argv[0]);
6   for (int i = 0; i < count; i++)
7      cout << argv[1] << endl;
8 }
```

The program should print out a number of copies of its second argument. You could put in a number of output statements to display the state of the variables, and that would certainly do the job in this case. However, it is tedious to "wire up" a large program in this fashion, since it's hard to decide what variables not to print out!

The `#b` Command

Alternatively, you can set a breakpoint at line 6, by using the `#b` command, and then you can run the program with some command-line arguments, as in the following example:

```
;> #b 6 main.cpp
breakpoint 0 set
;> #r 10 fred
halted at  main 5,main.cpp
```

When you run the program, the system gives a message that it has stopped at line 6. The program's execution is frozen, and you can access all the local variables that are in scope at that point (that is, `argc`, `argv`, and `count`). After the program has halted, you can look at the values of these variables by typing their names, or any expression involving them (please note that the comments between square brackets are not part of the session):

```
;> count;
(int) count = 0           [much as expected. But why?]
```

```
;> argv[0];                [what was the argument of atoi?]
(char *) "main.cpp"        [oops! This is the script name]
;> argv[1];                [and this should be the number...]
(char *) "10"
```

The problem has become obvious: You have forgotten that `argv[1]` is the first argument, not `argv[0]`. Because you are so accustomed to C++ arrays beginning at zero, this is a common error. (No nontrivial program works the first time, so don't worry about finding and needing to fix problems like this.)

To continue execution of a halted program, use `#r` with no parameters.

When you are finished with a breakpoint, you use the same command to toggle the breakpoint; that is, if the breakpoint was set, it is removed. Here is `linear_pos()` from Chapter 3, "Arrays and Algorithms," again showing line numbers just for illustration. A breakpoint is set inside the `for` loop:

```
30 int linear_pos(int arr[], int n, int val)
31 {
32 for(int i = 0; i < n; i++)
33   if (arr[i] > val) return i;
34 return -1;
35 }

;> #l insert.cpp
;> #b 33
breakpoint 33 set
;> int arr[] = {1,3,9,11,15,0};
;> linear_pos(arr,6,9);
halted at  int linear_pos(int*,int,int) 33,insert.cpp
;> i;
(int) i = 0
;> #r
halted at  int linear_pos(int*,int,int) 33,insert.cpp
;> i;
(int) i = 1
;> arr[i];
(int) 3
;> #b 33
breakpoint 33 unset
;> #r
(int) 3
```

After you are satisfied with the operation of the loop, then the command `#b 33` switches off the breakpoint at line 33, and the function can complete its execution.

Temporary Breakpoints: The #gt Command

UCW also supports temporary breakpoints, which automatically remove themselves when the program halts. The command is "go to" (#gt) but otherwise works exactly like #b. Continuing the preceding example:

```
;> #gt 33
breakpoint 33 temp set
;> linear_pos(arr,6,3);
halted at  int linear_pos(int*,int,int) 32,insert.cpp
;> i; arr[i];
(int) i = 0
(int) 1
;> #r
(int) 2
```

Note that it is not necessary to turn this breakpoint off; this automatically happens when execution resumes.

Inspecting Values

When you inspect your program's local variables, you can also evaluate any expression that involves those variables. You can also call any of the program's functions, except for the function you are currently debugging.

Writing Debug Code

UnderC lends itself to a development style in which you work on simple functions and test them individually; you end up with a set of operations you can trust that are convenient for the programs you are writing. You should write some functions specifically to help you debug your program (for example, code that dumps out the contents of a list in comprehensible form or performs sanity checks on data structures). This takes care and effort, but it nearly always makes a difference in a big project.

For instance, when I was developing UnderC, I wrote code to display the pcode generated by the compiler, a dissembler. (A *dissembler* is the opposite of an assembler; it reads raw program memory and writes it out as machine instructions.) I did not expect users to be interested in the pcode, but it was absolutely essential in getting the compiler debugged, and it was completely worth the hundred or so extra lines of code.

Often it is not necessary to write extra code because the preprocessor can be used to set up shortcuts. For example, the following code calls a function repeatedly and inspects a set of local variables; you can save yourself a lot of typing by using #define at each break:

```
;> #define W a; i; k;
;> #b 12 fred.cpp
;> call_fred(2.2,1,1);
Stopped at 12 fred.cpp
;> W
(double) a = 1.2
(int) i = 1
(int) k = 2
;> #r          [resume execution]
```

Note that you could not define a function to do this because functions cannot be nested within other functions.

A very useful command is `#lv`, which will list all the local variables in the current function scope, if you have halted within a function.

Modifying the Program while It's Running

You can modify the debugged program; you can change a variable's value if you know what it should be. But be careful with declarations when you are stopped within a function. While you are stopped, you are actually in the function scope. Anything you then declare in interactive mode falls within that scope and is inaccessible when you leave the function.

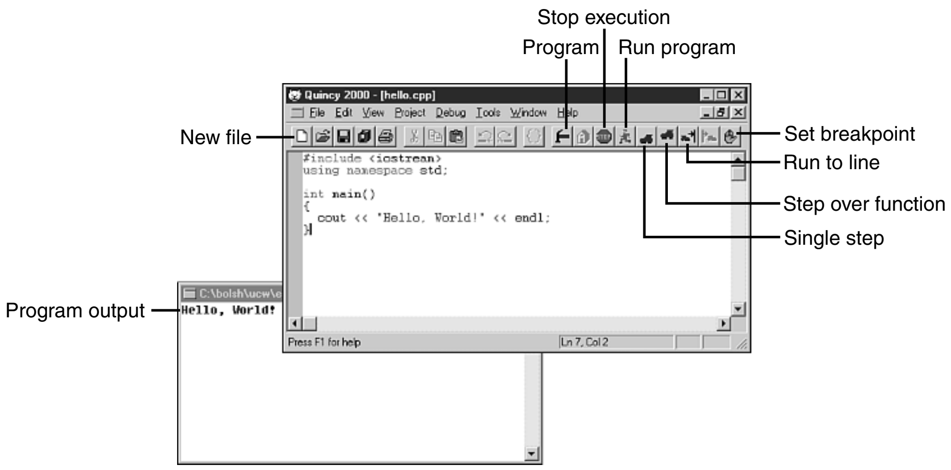
While debugging, you can modify and recompile files other than the one you are working on. This is very useful in larger programs and a good reason to break code up into separate files. Traditional systems require you to stop the program first and then rebuild.

Unfortunately, at the moment you need to reset breakpoints after a recompile. The reset breakpoints command (`#bs`) has a slightly different syntax: first it includes the file and then a list of the new line numbers (in order) for any breakpoints in that file. UCW cannot keep track of modifications to a file, and those modifications might move breakpoints to different places.

Using Quincy 2000

It is usually more convenient to use an integrated development environment than to issue commands yourself. For this book I have modified Al Stevens' excellent Quincy environment to work with UnderC. Quincy also works with free compilers like GCC and BCC55. It saves you from having to remember (and type) the commands yourself, and it keeps track of breakpoints. UCW runs as a separate program, and Quincy sends it # commands. If, for some reason, UCW gets confused (which happens occasionally), you can shut it down and restart it separately from Quincy, and Quincy will pick up where you left it.

Figure A.1 shows Quincy 2000 in action, indicating the meaning of the most important icons on its application toolbar.



```
#include <iostream>
using namespace std;

int main()
{
  cout << "Hello, World!" << endl;
}
```

```
Hello, World!
```

Figure A.1: *Most important operations using Quincy 2000.*

Entering a Program into Quincy

Here is how you create a simple C++ program with Quincy:

1. Create a new document in Quincy by selecting File, New, by pressing Ctrl+N, or by clicking the first icon on the Toolbar.
2. Specify “C++ Source File” in the list of choices.
3. Quincy creates a new file called `noname1.cpp`. In this window, type the usual “hello world!” program, as shown:

   ```
   #include <iostream>
   using namespace std;

   int main()
   {
     cout << "Hello, World!" << endl;
   }
   ```

4. You can now save this with File, Save, by pressing Ctrl+S, or by clicking the second icon.

NOTE

Syntax highlighting shows the C++ keywords, the variables, comments, and string literals in different colors. This can be very useful for visually detecting problems (for example, forgetting to finish a comment or a string).

Running a Program with Quincy

To run the program, follow these steps:

1. Select Debug, Run, press F9, or click the “running man” icon.
2. Quincy automatically does three things: It saves the file (if the file was changed), it compiles the file by using `#l`, and it runs the program by using `#r` and any command-line arguments.
3. A separate program window that contains your program output appears within milliseconds. If there was a compile error or a runtime error, you are told about it and are immediately placed at the appropriate line, which saves you a lot of labor. If a program gets out of control, clicking the stop icon causes the `#s` command to be sent to UCW.

Specifying Command-line Arguments

To specify the command-line arguments for program execution, follow these steps:

1. Select Tools, Options, and you will get the Options dialog box.
2. Select the Run tab and then type in any command-line arguments, or you can choose to let Quincy prompt you for these arguments each time the program is run.

Switching between UnderC and GCC

In the Options dialog box, note the “Run UCW” option; if you switch it off (it is on by default) and run the program as you did before, the program is compiled and linked with GCC. The progress is indicated in the status bar at the bottom right of the main window; if there are any errors, a list of errors appears, and you can click the list to inspect.

A shortcut for switching between UCW and GCC is Options, Use UCW. This menu item will be checked if you are currently using UCW. (You can also toggle this setting with Ctrl-U.)

Setting Breakpoints with Quincy

You set breakpoints by using the “hand” icon on the Toolbar (a shortcut is function key F2), and these appear as a small hand icon in the left-hand margin of the editor window. When the program stops, this icon changes to show that execution has halted at that point.

The two stepping operations (controlled by the “boot” icons) are not available for UCW, but are for GCC. Single-stepping means going through the program one line at a time; the difference between the two operations is that the first will go into a function definition, and the second will "step over" (hence the "raised boot" icon) the function.

The Watch Window

The watch window (which you open by selecting Debug, Watch or by pressing Ctrl+W) lets you set a number of variables (or expressions) that you can inspect. This works slightly differently in UCW mode than when working with GCC, because in UCW mode it is possible to inspect variables when the program is running, but not halted. You can also inspect more complicated expressions in UCW mode because the full power of the UCW expression compiler is available.

You can always switch to the UCW window and work from there; this gives you the flexibility to call specific functions, and so on.

B

A Short Library Reference

Most of the power of a programming language comes from its libraries, which give a programmer access to hundreds of thousands of lines of well-debugged code. The C/C++ library is particularly rich and often supplies more than one way to handle a situation; knowing the standard library well keeps you from having to reinvent the wheel. Conversely, any language that can't be extended by libraries will not last, as illustrated by the case of Pascal. Standard Pascal died out because it did not offer powerful libraries, but Pascal has been revived in recent years, thanks to Borland making the Delphi VCL library comprehensive and extensible. Generally, you can find excellent C++ libraries freely available to do specialized tasks; examples are the Visualization Toolkit (VTK) and SAX and DOM libraries for dealing with eXtensible Markup Language (XML).

NOTE

The standard library does not currently have any classes for graphical user interfaces (GUIs). One of the reasons for Java's popularity is precisely because it has a standard GUI framework.

Whole books have been written about the C++ libraries (Stroustrup's book *The C++ Programming Language*, 3rd edition, Addison-Wesley 1997, devotes more than 300 pages to it), so this appendix does not try to cover everything. Rather, like a good librarian, it indicates what's important and helps you navigate the online documentation. This chapter also does not give overloaded versions of all the methods and functions; rather, it discusses only the main methods and functions that show the range of functionality offered by the libraries.

The UnderC standard "pocket" library is still under development, and there are some classes and standard algorithms that are not finished yet. So currently there are no `valarray` or `complex` classes; updates will be available. But all of these classes are available from either GCC or BCC32.

The <iostream> Library

Input/output (I/O) in C++ relies on overloading the operators << (for writing) and >> (for reading). All output classes derive from `ostream`, so overloading the >> operator for `ostream` means that you can use that operator for any output class. It is important to realize that file output is *buffered*; that is, the data is not immediately written to the actual disk file. This is done to speed up I/O because disk access occurs much more slowly than memory access. You can flush file streams (that is, write them to disk) by using the `ostream::flush()` method when the streams are closed. This is why it is important that file streams are closed properly.

The standard streams are `cin`, `cout`, and `cerr`. (`cerr` is used for error output. It will also appear on the screen, but is unbuffered, so no output is lost.) `<iostream>` also defines some special objects, called *manipulators,* that modify the stream; an example is `endl`, which inserts a newline and flushes the stream.

Reading Data

Data items are read by using `operator>>`, which usually ignores whitespace (including newlines). You call `eof()` to determine whether a stream is finished, and you call `bad()` to determine whether an error has occurred. If `in` is an input stream, then `! in` is `true` if the stream is bad or past the end of the file; you can test `in >> var` directly, as follows, because `istream` has a conversion operator to integer:

```
while (in >> x >> y) diff += (x - y);
```

Reading Character Strings

The extraction operator (>>) is overloaded for both standard `strings` and plain arrays of characters. In both cases, the next *token* (that is, text delimited by whitespace) is read in; reading into a `string` is safer than reading into an array because `strings` do not overflow (they are resizeable); however, raw character data is faster than `string` data. (Safety versus speed is a common choice in programming, and C++ offers both.) If you want to read a whole line of text, you can use the two main versions of `getline()`. The second form of `getline()` is implemented as a function, not as a method, and it takes a standard `string` argument.

```
istream& istream::getline(char *buff, int sz);
istream& getline(istream& is,string& s);
```

Both versions of `getline()` return a reference to the input stream, which can be used in two ways. First, you can chain calls, as shown in the following example. Second, you can test the return value, relying on the conversion operator:

```
;> const int BSIZE = 256;
;> char b1[BSIZE],b2[BSIZE];
;> string s;
;> cin.getline(b1,BSIZE).getline(b2,BSIZE);   // two lines are fetched
one
two
;> ifstream in("temp.txt");
;> while (getline(in,s)) cout << s << endl;   // while in.eof() isn't true
```

`<iomanip>` defines the `setw()` manipulator, which can be used as follows to force the input width, if you need to read the next `n` characters:

```
// stream contains 'dolly the sheep'
in >> setw(4) >> s; // s contains 'doll'
```

Reading File Streams

You can read files by using the `ifstream` class, which is derived from `istream`, and you can write files by using the `ofstream` class, which is derived from `ostream`. You use `open()` to access a file; it returns `false` if the operation failed. You should always check whether the operation failed. The file needs to be closed at the end, and the class destructor does this. Note that `open()` does not currently take a `string` argument, so you have to use `s.c_str()` to get the underlying character data if you are using standard `strings`.

The file stream `open()` method can take an extra argument, which controls how the file is opened. This table shows the main `ios` flags and their meanings; they may be combined using `|`:

```
ios::in         -file opened in input mode
ios::out        -file opened in output mode
ios::app        -file append; any writing starts at end of existing file
ios::binary     -file opened in binary mode; no text translation
ios::in | ios::out       -both read and write!
ios::app | ios::binary   -append to binary file
```

Formatting Output

You control floating-point precision with the `setprecision()` manipulator; you control the width of the output field with `setw()`, as in the following code; please see `stdlib\test-io.cpp`:

EXAMPLE

```
void test_out()
{
  int val = 648,k;
  double pi = 4*atan(1.0); // a way to calculate PI...

  // note that it is necessary to  switch back to decimal....
  cout << "Hex " << hex << val << " Dec " << dec << val << endl;
  // floating-point precision can be set using setprecision
  cout << "Default precision   " << pi << endl;
  cout << "Precision(12) " << setprecision(12) << pi << endl;
  cout << "Precision(6)  " << setprecision(6) << pi << endl;
  cout << "Precision is (" << cout.precision() << ") " << endl;

  // can control the width using the setw() manipulator
  // note that the width only applies to the _next_ field!
 for(int i = 0; i < 5; i++)
     cout << setw(10) << i*pi;
 cout << endl;


// setting formatting flags: this forces positive numbers to be
// displayed with an explicit plus sign.
 cout.setf(ios::showpos ) ;
 for(int i = 0; i < 5; i++)
    cout << 10*i << ' ';
 cout << endl;

}
```

OUTPUT

```
Hex 288 Decimal 648
Default precision   3.14159
Precision(12) 3.14159265359
Precision(6)  3.14159
Precision is (6) 3.14159
         0   3.14159   6.28319   9.42478   12.5664
+0 +10 +20 +30 +40
```

The C++ Standard `string` Class

A standard `string` is essentially a resizable array of characters, with methods to perform substring searching and extraction, and overloaded operators to construct longer strings by concatenation. In the following table, `cstr` is an argument of type `const string&`, `ch` is an argument of type `char`, and `i` and `n` are arguments of type `size_type` (which is usually an unsigned long):

```
string();       // default constructor - empty string ("")
string(cs);     // C++ string from a C character string
string(n,ch);   // ch repeated n times.
int size();     // number of characters
int length();   // ditto
int capacity();  // actual space available
void reserve(n); // make space available
string substr(i,n);   // n chars from  i (zero-based index)
int find(cstr);     // first cstr (npos if not found)
int rfind(cstr);    // last cstr
int find(ch);       // ditto, but for characters...
int rfind(ch);
void replace(i,n,cstr); // n chars after i replaced by cstr
void append(cstr);          // append cstr to the end
string& operator += (cstr);  // ditto
string& operator += (ch);    // ditto for characters
bool operator==(cstr1,cstr2);
string operator+(cstr1,cstr2);
```

The special constant `string::npos` is an unsigned value that is larger than any valid string length; it is –1 as a signed integer, which is why you occasionally see this value used. All the `find()` routines return `npos` on failure; if `npos` is used as a length (as in `replace(2,string::npos,"help")`), it means "to the end of the string."

The key to building up strings efficiently is to note that `std::string` overallocates. The string's `capacity()` method is always greater than the `size()` method. You can explicitly ask for storage to be allocated by using `reserve()`. This prevents possibly expensive resizing operations, which usually involve copying. The following function shows the difference between `size()` and `capacity()` (code found in `stdlib\test-string.cpp`).

```
void test_string()
{
 string s;
 s.reserve(200);
 cout << "size = " << s.size()
      << " capacity = " << s.capacity() << endl;
 for(int i = 0; i < 80; i++) s += '*';
 cout << "size = " << s.size()
      << " capacity = " << s.capacity() << endl;
}
size = 0 capacity = 200
size = 80 capacity = 200
```

You can get a character pointer from `std::string` by using `c_str()`. You should not keep this pointer because it usually becomes invalid when the string is destroyed.

C++ Standard Containers: `list` and `vector`

All the standard containers like `list` and `vector` are class templates and have the following methods in common. That is, they all have operations for adding or removing elements at the end (`push_back()` and `pop_back()`), and you can access the first and last elements directly. They also have an iterator type, which refers to elements and can be used to access each element in turn (that is, to iterate through the container.) The standard methods begin() and end() return iterators to the first element and just past the end respectively. Although string is not a full container, it also supports these methods.

```
size_type size()   // number of elements
type      back()   // last element
void      push_back(type val);  // add an element to the end
void      pop_back();       // remove an element from the end
type      front();  // first element
iterator  begin();  // iterator at beginning of container
iterator  end();    // iterator pointing just past the end
```

The common operations allow you to use the containers in similar ways. The first four operations allow you to use the containers as stacks, and the iterators can be used to iterate through the elements. This common interface allows a user to write code using containers without having to know precisely what container is being used; this makes the standard algorithms possible. To avoid excessive copying, container objects are usually passed as references.

The Standard Vector: `<vector>`

std::vector is meant to be a fast, resizable array, which you access by using operator[]. You can build up this type of array one element at a time by using push_back(). A detailed discussion of std::vector can be found in Chapter 3, in the section "Resizable Arrays."

Resizing involves dynamic allocation and copying, and it can be slow; therefore, std::vector usually allocates more memory than is needed. capacity() returns the actual space available, and size() returns the current number of elements. That is, capacity() is always greater or equal to size(). If you know in advance how many elements there are, it is a good idea to allocate space up front by using reserve(). This example shows the difference between size(), resize() and capacity(), reserve():

```
;> vector<int> vi;  // size zero!
;> vi.reserve(100);
;> vi.capacity();
(int) 100
;> vi.size();
(int) 0
;> vi.push_back(648);
;> vi.size();
(int) 1
;> vi.resize(5);
;> vi.size();
(int) 5
;> vector<double> fd(100);  // size 100, capacity 100+x
;> for(int k=0;k<100;k++) fd[k] = k;
```

The Standard Deque: `<deque>`

`std::deque` (pronounced ‘deck’) has the same interface as `std::vector`, except that it has efficient operations on the front as well as on the back. (Having operations on the front would be inefficient for a vector; that is why they are not provided.) These methods allow you to add elements to the front of the deque, and to remove them:

```
void push_front(T val);     // add to front
void pop_front();           // remove from front
T front();                  // value of front
```

`std::deque` is useful for implementing first in, first out (FIFO) queues. You can use `push_front()` to put something in the queue, and you can use `back()` and `pop_back()` to take something out of the queue.

The Standard List: `<list>`

`std::list` has the same basic interface as the other containers—that is, `size()`, `begin()`, `end()`, `back()`, `push_back()`, and so on. It also has operations on the front of the list, like `std::deque`: `push_front()`, `front()`, `pop_front()`.

`std::list` differs from `std::vector` in that it does not have random access (`operator[]`), it allocates only as much memory as necessary, and it is easier to insert, delete, or reorder elements. See the section “Lists” in Chapter 3.

The following useful `std::list` operations are particularly efficient for linked lists:

```
void insert(iterator lp, const T& val);
void erase (iterator lp);
void insert(iterator lp, iterator is, iterator ie);
void erase (iterator is, iterator ie);
void splice(iterator lp, list& ls);
void splice(iterator lp, list& ls, iterator is, iterator ie);
void merge(list& ls);
void remove(const T& val);
void unique();
void sort();
```

`insert()` and `erase()` can be also be used with `vector`, but they are more efficient with lists. The idea is that `insert()` will put the new element before the specified position; `erase()` will remove the element at that position. For example, if `ls` is a `list<int>` and `v` is a value, then:

```
// insert in front of list (same as ls.push_front(v))
ls.insert(ls.begin(),v);
// insert at end of list (same as ls.push_back(v))
ls.insert(ls.end(),v);
```

```
// insert 67 before the value 42 in the list
list<int>::iterator li = find(ls.begin(),ls.end(),42);
ls.insert(li,67);
// erase the value 42 from the list
ls.erase(li);
// erase the first element (same as ls.pop_front())
ls.erase(ls.begin());
// erase the last element (same as ls.pop_back())
ls.erase(ls.end());
// erase all elements of list (same as ls.clear())
ls.erase(ls.begin(),ls.end());
// insert another list at the front of our list
list<int> l2;  l2.push_back(10); l2.push_back(20);
ls.insert(ls.begin(),l2.begin(),l2.end());
```

The `splice()` and `merge()` operations work with another list. `splice()` works rather like the full version of `insert()` shown in the last example; the difference is that it actually moves the elements from the source list. There is no actual copying involved. This shows the two ways to insert another list into a list; `splice()` is more efficient but does clear out the source.

```
  // insert l2 before the value 648 in the list
  li = find(ls.begin(),ls.end(),648);
  ls.insert(li,l2);      // l2 is not affected
  ls.splice(li,l2);      // l2 is empty!
```

`remove()` takes out elements with a specified value, and `unique()` eliminates duplicates, provided that the list is in order. To get a list into ascending order, use the `sort()` method.

C++ Standard Algorithms: `<algorithm>`

Template functions allow many common operations to be written once for different types. The standard algorithms all operate on sequences, rather than containers (see the section "Finding Items" in Chapter 3 for the reasons why). For example, to specify all elements of a container `c` you can use `c.begin(),c.end()`. Because the containers have the same basic interface (for instance, they all have `begin()` and `end()`) it is possible to write code that will work whether you are using `list`, `vector`, or any other standard container.

Many of the standard algorithms are very straightforward template functions. Most of them are simple loops. Here is a implementation of `for_each()`, `copy()`, and `transform()`:

```
template <class In, class Fn>
  void for_each(In i1, In i2, Fn fn)
  {
    while (i1 != i2) {
      fn(*i1);
      ++i1;
    }
  }
template <class In, class Out, class Fn>
  void copy(In i1, In i2, Out oi)
  {
    while (i1 != i2) {
      *oi = *i1;
      ++i1; ++oi;
    }
  }

template <class In, class Out, class Fn>
  void transform(In i1, In i2, Out oi, Fn fn)
  {
    while (i1 != i2) {
      *oi = fn(*i1);
      ++i1; ++oi;
    }
  }
```

The standard algorithms can be used to replace loops in any case where you access data with iterators, such as where `++in` moves the iterator to the next value and `*in` accesses that value. Iterators are meant to be generalizations of C pointers, and because of how input sequences are specified, you can apply algorithms to ordinary arrays as well. The last iterator value is assumed to be the one just past the end, which would be `end()` for a standard container and `arr+`*n* for an array with *n* elements. Using the standard algorithms is usually as fast as creating a loop manually and explicitly because the code is inlined (unless otherwise specified).

For example, both `for_each()` and `transform()` apply a function to each element in a sequence. However, the function given to `transform()` has to return a value. This value will be written out to an output iterator, which can be refer back to the input. (See the second use of `transform()` that follows.) `transform()` is like `copy()`, except with an extra function application. Here are some examples of how to use these algorithms:

```
;> char *ar[] = {"2","5","7"};  // an array of C strings
;> for_each(ar,ar+3,puts);
2
5
7
;> vector<int> v(10);
;> transform(ar,ar+3,v.begin(),atoi);
;> v[0]; v[1]; v[2];
```

```
(int&) 2
(int&) 5
(int&) 7
;> double ad[3];
;> copy(ar,ar+3,ad);  // copy from int array to double array
;> transform(ad,ad+3,ad,sqrt);  // transform in-place!
;> ad[0];
(double) 1.41
;> int sum = 0;
;> void sum_it(int i) { sum += i; }
;> for_each(v.begin(),v.end(),sum_it);
;> sum;
(int) 14
```

A useful thing about the standard algorithms is that they are quite happy working with mixed kinds of data, such as moving an integer array into a double array with `copy()` or moving integers from an array into a `vector<int>`. Please note the standard way to apply a function to all elements of a sequence: `transform(ad,ad+3,ad,sqrt)` applies `sqrt()` to all three elements of the array `ad`.

The last part of the example calculates the sum of a `vector`'s elements and works well, but `accumulate()` (discussed later in the section "Standard Numerical Algorithms") does the same job more neatly, and without global variables.

The algorithms that generate output do not do any range checking. You either have to make sure that the output is large enough or create space as you go along, by using `back_inserter()`:

```
;> list<int> ls;
;> vector<int>::iterator vii;
;> for(vii = v.begin();vii != v.end();++vii) ls.push_back(*vii);
;> copy(v.begin(),v.end(),back_inserter(ls));
```

This example shows two ways of adding the elements of a `vector` to a `list`. The first way does it with an explicit loop over all elements of the `vector`, using `push_back()` to add each element. The second way is to use `copy()` with `back_inserter(ls)` as the target; this way is just as fast and involves less tricky typing. `back_inserter()` in fact uses `push_back()`, so it can only be used with types that support this. (See the section "Template Functions That Generate Template Classes" in Chapter 10 for a simple implementation of `back_inserter()`.)

Searching and Finding

In all the following tables of algorithms, `In` means any input iterator, `Out` means any output iterator, `P` a predicate function (that is, a function that returns `true` or `false` for an element), and `T` is any type.

The input sequence (i1,i2) means everything from i1 up to but not including i2. It is very common to use the format c.begin(),c.end(), for example, to operate on an entire container c. In interactive work, you can define a macro ALL() using #define ALL(x) c.begin(),c.end(), and thereafter say things like find(ALL(ls),42). This saves typing when you are exploring the algorithms but is not always considered good style in production code.

```
// return an iterator to first position of val
In find(In i1, In i2, T val);
// like find, except for val for which pred is true.
In find_if(In i1, In i2, P pred);
// In find_first_of(In i1, In i2, In2 s1, In2 s2);
// In adjacent_find(In i1, In i2);
// If (i1,i2) is a sequence in order, then find val
In binary_search(In i1, In i2, T val);
// sort the input sequence
void sort(In i1, In i2);
```

The find() family of algorithms is convenient for finding all instances of a value, by using the result to define the input sequence for the next search, as in the following function. (Please note the const_iterator used because the function is passed a const reference.)

```
void print_all(const list<int>& ls, int val) {
  list<int>::const_iterator li = ls.begin(), lend = ls.end();
  while ((li = find(li,lend,val) != lend)
       cout << *li << endl;
}
```

Any algorithm with a name that ends in _if takes a *predicate*, which is a function (or function-like object) that returns true or false. In the following example, find_if() is used to find the next alphabetic character in the string s.

adjacent_find() finds the first repeated value, and find_first_of() compares values of one sequence to see whether they are in another sequence. In this example, the first sequence is some arbitrary text, and the second sequence is the set of vowel characters. find_first_of() will then find the first position in the text where there is a vowel. Again, you need to use operator* to get the actual character at that position:

```
;> string s = "404 Not Found";
;> string::iterator is = s.begin();
;> is = find_if(is,s.end(),isalpha);
;> *is;
(char) 'N'
;> int arr[] = {2,3,6,6,8,9};
;> *adjacent_find(arr,arr+6);
(int) 6
```

```
;> char txt[]="quick fox, brown dog";
;> char vowels[]="aeiou";
;> *find_first_of(txt,txt+strlen(txt),vowels,vowels+5);
(char) 'u'
```

The `find()` algorithms use a simple linear search, which is generally the only type of search you can do on arbitrary sequences, but `binary_search()` can do much better, if the sequence is sorted. See the section "Searching and Sorting" in Chapter 3 for more details.

Comparing and Counting

```
bool equal(In i1, In i2, In2 i3);
bool equal(In i1, In i2, In2 i3,Bin p);
pair<In,In2> mismatch(In i1, In i2, In2 i3);
pair<In,In2> mismatch(In i1, In i2, In2 i3,Pred p);
size_t count(In i1, In i2, T val);
size_t count_if(In i1, In i2, T val,Pred p);
```

The benefits of having algorithms that do comparisons are that you can compare very different kinds of data. For instance, you could compare a list of integers with an array of integers, and then `equal()` would be true if they contained the same numbers.

Using the second version of `equal()`, you can specify exactly what you mean by equality. In the following example, you can compare arrays of character pointers by using the classic C function `strcmp()`, which returns 0 if two pointers to refer to identical characters. To do this, you must define a function `cmp()` that returns true instead of 0. (Normal comparison would be between pointers, which would rarely be meaningful.) This little program will print out the message `"matched"` if it is given the command line `"one two three"`. I have deliberately left out the usual `using namespace std`, so you can clearly see all the library functions being used here:

EXAMPLE

```
// test-equal.cpp
#include <algorithm>
#include <cstring>
#include <cstdio>
bool cmp(char *p1, char *p2) { return !std::strcmp(p1,p2); }
char *reqrd[] = {"one","two","three"};
int main(int argc, char **argv)
{
  if (std::equal(argv+1,argv+argc, reqrd, cmp))
      std::puts("matched!");
}
```

`mismatch()` returns the precise place where two sequences differ, and it returns a pair of iterators. Pairs are probably the simplest classes in the standard library; they just contain two values of different types called `first` and `second`, as shown here:

```
template <class T1, class T2>
  struct pair {
    T1 first;
    T2 second;
    pair(T1 _first, T2 _second)
      : first(_first),second(_second)
    {}
  };
;> string s = "alice", vowels = "aeiou";
;> *(mismatch(s.begin(),s.end(),vowels.begin())->first);
(char) 'l'
```

Finally, `count()` does the common task of counting the number of times a given value appears in a sequence. The useful `count_if()` uses a predicate function to tell which elements to count:

```
;> string s = "the dog is outside";
;> count(s.begin(),s.end(),'o');
(int) 2
;> int arr[] = {23,4,-2,5,2,-1,-3,10};
;> bool less0(int v) { return v < 0; }
;> count_if(arr,arr+8,less0);
(int) 3
```

Filling and Generating Sequences

`fill()` copies a specified value into all elements of a sequence; `generate()` puts the result of a specified function into the elements. They both come in two forms: first, where the sequence is specified by (begin,end), and second, where it is defined by a starting point and a number:

```
// Fill (i1,i2) with val.
void fill(In i1, In i2, T val);
// Fill n values starting at i1 with val
void fill_n(Out o, Size n, T val);
// Generate values for (i1,i2)
void generate(In i1, In i2, Fn fn);
// Generate n values starting at i1
void generate_n(Out o, Size n, Fn fn);
```

These four algorithms are useful to initialize arrays and containers with initial values. `fill(ALL(v),0.0)` initializes the vector `v` to `0`, and `fill_n(arr,n,0.0)` does the same for an array. If you have a function that takes no arguments, you can use it as a generator, as in the following example:

```
;> int randint() { return rand() % 100; } // return value is from 0 to 99
;> generate_n(nums,20,randint);   // 20 pseudo-random numbers into the array nums
```

`generate()` can do very interesting things when you use a function-like object. The following defines a function-like class `nseq`, which generates a rising sequence, or "ramp." The example shows an explicitly created object of type `nseq` and shows that each 'call' of this object will produce the next integer in the sequence. Passing a `nseq` object to `generate_n()` can be used to initialize an array to `0..n`. `nseq` can be easily generalized to do any increment (integer or otherwise):

```
struct nseq {
     int icount;
     nseq() : icount(0) {}
     int operator() (){ return icount++; }
  };
;> nseq ns;
;> ns();
(int) 0
;> ns();
(int) 1
;> double a[10];
;> generate_n(a,10,nseq());  // 0,1,. ..9
;> list<int> ls;
;> generate_n(back_inserter(ls),20,nseq());   // using back_inserter() to add
0..9 to the list
```

Modifying Sequences

`replace()` will replace all occurances of some value with another value; `remove()` will remove all occurances of some value. `remove_if()` is similar, except it removes all values that satisfy some predicate function. For instance, the function `less0()` defined earlier, which returns true for values less than zero, can be used to remove all negative values from a sequence.

```
// replace v1 by v2 in the sequence (i1,i2)
void replace(In i1, In i2, T v1, T v2);
// remove val from a sequence
void remove(In i1, In i2, T val);
// remove val if we match the condition
void remove_if(In i1, In i2, Pred p);
// rotate sequence
void rotate(In i1, In mid, In i2);
```

Minimum and Maximum Values

You will very often need the minimum or maximum of a set of values. `min()` and `max()` apply to pairs of values, and `min_element()` and `max_element()` work with sequences of values. Note that the last two functions return an iterator to the value, not the value itself! This makes them more powerful, because you can then use the iterator to modify the sequence, but you will have to remember to use the dereference operator (*) to get the actual minimum or maximum value:

```
// Maximum/minimum of t1 & t2
T max(T v1, T v2);
T min(T v1, T v2);
// minimum/maximum element of sequence
In min_element(In i1, In i2);
In max_element(In i1, In i2);

// for example, the maximum value of a list ls
mx = *max_element(ls.begin(),ls.end());
```

Numerical Operations

C++ is particularly powerful at engineering and scientific applications, which require intensive numerical calculations. There is the full range of mathematical functions inherited from C, now including versions that work with complex numbers. There are standard algorithms defined in `<numeric>`, which do full and partial sums of sequences. There is also the `valarray` class, which is optimized for high-performance applications and works like a mathematical vector.

Mathematical Functions: `<cmath>`

All the mathematical functions take double arguments and return double values. (In addition, C++ makes complex versions available.) Be careful with functions like `sqrt()`, `log()`, and `pow()` that are not defined for all values of `x`; this is called a *domain error*. For example, it is an error to pass a negative real value to `sqrt()`.

When calculating angles from values of the tangent, you should use `atan2()` rather than `atan()` because it provides an answer that is not ambiguous; the answer is in the range `-pi` to `pi`, which is the full circle.

```
sin(x);   // sine of x (all angles in radians)
cos(x);   // cosine of x
tan(x);   // tangent of x
asin(x);  // arc sine of x (returns radians)
acos(x);  // arc cos
atan(x);  // arc tan
atan2(y,x); // arc tangent of y/x (better than atan(x)!)
pow(x,y);   // x to the power of y
sqrt(x);    // square root of x
log(x);   // ln(x) (natural logarithm)
log10(x); // log(x) (base 10 logarithm)
exp(x);   // exponential function; same as pow(e,x)
ceil(x);  // truncate upwards to nearest integer
```

```
floor(x);  // truncate downwards to nearest integer
fabs(x);   // absolute value of x
```

In addition to the hyperbolic trigonometric functions (such as `sinh()`), most systems have the various Bessel functions, although these are not part of the ANSI standard.

Standard Numerical Algorithms: <numeric>

The following are specialized for numerical data and are found in `<numeric>`. These algorithms all work on sequences of numbers; `accumulate()` will sum the values, but you must give an initial value. `partial_sum()` will generate the running sum (for example, (0,1,2,4) becomes (0,1, 2+1, 4+2+1)), and `adjacent_difference()` will give the differences. Applying `partial_sum()` and then `adjacent_difference()` will give you back the original sequence. Please look at the stock statistics case study at the end of Chapter 3, "Arrays and Algorithms," for examples of these algorithms in action.

```
// sum of the elements in (i1,i2)
T accumulate(In i1, In i2, T val);
// put running sum of elements of (i1,i2) into j
Out partial_sum(In i1, In i2, Out j);
// put differences between elements of (i1,i2) into j
Out adjacent_difference(In i1, In i2, Out j);
```

Complex Numbers <complex.h>

One of the reasons scientific programmers preferred FORTRAN to C was because FORTRAN explicitly supported complex numbers. The standard C++ library defines a complex type that can be just as efficient as C or FORTRAN code.

A complex number has a real part and an imaginary part, and it is essential in most parts of science and engineering because it is the most general form of number; it's no exaggeration to say that normal floating-point real numbers are a special case of complex numbers. For example, here is an example program demonstrating operations on C++ complex numbers.

EXAMPLE

```
// test-complex.cpp
#include <iostream>
#include <complex.h>
using namespace std;

typedef complex<double> Complex;
const double PI = 3.14123;

int main()
{
 Complex x(0,1);               // the square root of -1
 Complex y = x*x;               // sure enough, y is now (-1,0)
 cout << "y = " << y << endl;  // can write (or write) as numbers
 y = sqrt(x);                  // can apply the usual functions
 x = pow(y,2.0);
 y = sin(x);
 x = conj(y);                 // the complex conjugate of y
 y = polar(1.0, PI/2);        // a complex number of length 1, argument pi/2
 x = 2.0;                     // can initialize with a double
// Can access the real or imaginary parts
 cout << x.real() << ' ' << y.imag() << endl;
// Or in polar coordinates: a length and an angle.
 cout << abs(y) << ' ' << arg(y) << endl;
}
```

This produces the following output:

OUTPUT

```
y = (-1,0)
2 1
1 1.57062
```

The usual operations (that is, `-`, `+`, `*`, `/`, `==`, `!=`) are available, as are the standard scientific functions (that is, `sin()`, `cos()`, `tan()`, `sinh()`, `cosh()`, `tanh()`, `exp()`, `log()`, `log10()`).

The `valarray` Class: `<valarray>`

The `valarray` type is like `vector`, but it is designed for fast arithmetic operations. It is in fact very much like a mathematical vector; you can multiply it by a scalar (that is, a single real number), add a `valarray` to another, apply mathematical functions, and so on, as in the following example:

```
#include <valarray>
...
#define FOR(i,n) for(int i = 0; i < n; i++)
valarray<double> v1(20), v2(20), v3(20);
FOR(i,20) v1[i] = i
v2 = sin(v1);
v3 = 2.0*(v1 + v2);
double arr[20];
copy(&v1[0],&v1[20],arr);
v1.resize(100);
FOR(i,100) v1[i] = 1.0/(1.0+i);
v1.apply(sqrt);                 // same as v1 = sqrt(v1)
```

`valarray` is convenient because you can use it in mathematical expressions, such as `v1+v2` or `sin(v1)` in the preceding code. These operations are shortcuts that apply to every element in the arrays. In fact, there is a method called `apply()` that applies a function to each element in turn. In the preceding code, `v1.apply(sqrt)` does the same as `v1=sqrt(v1)`, but is more efficient because the second expression involves a temporary array that is then copied. As with `strings`, you should be aware that temporary creation takes place in `valarray` expressions, and this can make them rather slower than explicit loops, so it's better to use `*=` than `*`, and so on.

`valarray` is only efficient for larger numbers of elements. An example where `valarray` would not be a good solution is using them as three-dimensional vectors in a graphics program that manipulates a lot of vectors. In this case it would be worthwhile writing your own Point class.

`valarray` is an old-fashioned class that predates the Standard Template Library (STL) and is even considered a bit of a dinosaur in some circles. The `valarray` class does not cooperate well with the standard algorithms because `valarray` does not have the usual container interfaces, such as `begin()` and `end()`. However, in the previous example, you could still use `copy()` to move `v1` into the array `arr`.

`valarray` has a `resize()` member function, but `valarray` is not a true resizable array because `resize()` does not preserve the old contents by copying. Be careful with the following constructors:

```
valarray ();         // these are
valarray (int sz);   // like vector!
valarray (T value, int sz);   // value comes before size!
```

Despite its deficiencies, `valarray` is designed to be fast, and it can be very useful when you need to crunch numbers.

You can access subsets of `valarray` by indexing it with `slice`, which specifies a sequence. The `slice_array` type behaves like `valarray`, but it really is a special alias that refers to an original `valarray` and hence cannot be copied directly. Because no copying is involved, it is very efficient. The following example prints out the odd elements of `v1`. This is done by indexing with a slice that begins at index 1, which has 10 elements and skips 2 elements each time. So `slice(0,10,2)` would refer to the even elements of `v1`:

```
FOR(i,20) v1[i] = i;   // 0,1,2,. ..
slice_array<double> v_odd = v1[slice(1,10,2)];
FOR(i,10) cout << v_odd[i] << ' ';  // 1.0, 3.0, 4.0 . . .
```

There are two other ways to extract subarrays. The first is to use a mask array, and the second is to use an indexing array. This example program shows how both of these methods can be used to extract a selected set of the values. A mask array is a `valarray<bool>`, which can be generated from a boolean expression; in this case, `v1 > 0`. I have dumped out the elements of this mask array, and you can see that it is `true` whenever `v1` is positive. That is, `mask[i]` is `true` if `v1[i] > 0`. Using this mask to index `v1` gives a `valarray<double>`, which only contains the positive values of v1. The second method of extracting values is to set up an indexing array, which must be of type `valarray<size_t>`. The result has the same size as the index and contains the specified values:

EXAMPLE

```
// test-valarray.cpp
#include <iostream>
#include <valarray>
using namespace std;

template <class T>
  void dump(const valarray<T>& v) {
   for(int i = 0; i < v.size(); i++)
     cout << v[i] << ' ';
   cout << endl;
  }

typedef valarray<double> DV;

int main()
{
 DV v1(15);
 int i;
 for(i = 0; i < 15; i++) v1[i] = sin(i/2);
 valarray<bool> mask = v1 > 0.0;
 dump(mask);

 DV gt_zero = v1[mask];                  // index with a mask

// gt_zero now contains all elements of v1 which are > 0;
// Calculate the sum of v1[1]+v1[4]+v1[10]
 valarray<size_t> index(3);
 index[0] = 1;
 index[1] = 4;
 index[2] = 10;
 DV subset = gt_zero[index];           // index with an index array;
 dump(subset);
 cout << "sum of these 3 elements " << subset.sum() << endl;
}
```

Here is the output of this program:

OUTPUT

```
0 0 1 1 1 1 1 1 0 0 0
0.841471 0.14112 0
sum of these 3 elements 0.982591
```

C Library Functions

Although many of the C library functions are no longer necessary in modern C++ code, you will probably see them in older code. In general, the C library header `<name.h>` is `<cname>` in C++, and the functions appear in the std namespace. Some parts of the C library, such as the <cctype> routines and the mathematical functions, remain as essential as always.

Character Classification: <cctype>

The following are fast and portable ways to classify characters, and they return nonzero if they are true. Their use is discussed in Chapter 4, in the section “Classifying Characters.”

isalpha(ch)	islower(ch) or isupper(ch)
isalnum(ch)	isalpha(ch) or isdigit()
isdigit(ch)	'0'..'9'
isxdigit(ch)	hexadecimal digit
islower(ch)	lower-case letter
isupper(ch)	upper-case letter
isprint(ch)	printable character, including space
ispunct(ch)	punctuation
isspace(ch)	whitespace; (space,newline,tab,carriage return)

The following are two case-conversion functions:

```
int tolower(int ch);
int toupper(int ch);
```

The C String Functions: <cstring>

String functions were common in C++ code before a standard string class was agreed upon. A C string is a pointer to a block of characters that ends with the null character, 0, which is written \0; the length of the string does not include this \0. Note that the copying operations strcpy() and strcat() work from right to left, and they assume that the target buffer is big enough to accept the source string. You are completely responsible for making sure that your strings do not overflow, and resizing them involves reallocating and then copying. These operations are done quietly by std::string, which is safer and can sometimes even be faster than the C string functions.

In the following declarations, s and t are of type char *, and cs and ct are of type const char *; n is of type size_t; and c is a character. Searching functions (that is, strchr(), strrchr(), strstr()) return NULL if they are unsuccessful:

```
char *strcpy(s, ct);   // copy ct to s, including last '\0'.
char *strncpy(s,ct,n); // copy at most n chars from ct to s
int   strlen(cs);      // length of cs, not including '\0'
char *strcat(s,ct);    // append ct at the end of s
char *strncat(s,ct,n); // append at most n chars of ct to s
int   strcmp(cs,ct);   // compare cs to ct; return 0 if equal
char *strchr(cs,c);    // pointer to position of c in cs
char *strrchr(cs,c);   // pointer to last position of c in cs
char *strstr(cs,ct);   // ditto for a substring ct
char *strtok(s,ct);    // extract tokens delimited by ct
size_t strspn(cs,ct);  // length of beginning sequence in cs
                       // consisting of characters in ct
size_t strcspn(cs,ct); // ditto, but characters not in ct.
char *strpbrk(cs,ct);  // pointer to first occurrence in cs of
                       // any character from ct
```

The key to manipulating C strings is knowing how to manage C pointers. Pointers are basically nonconstant arrays, so you can use array subscripting to access individual characters. Adding an integer to a pointer gives a new pointer; `*p` gives the character at `p`, and `++p` increments to the next character. Notice the complete correspondence to C++ iterator notation; in fact, iterators are a generalization of pointers. This example shows how character pointers can be manipulated; they are array-like and can be indexed (note that `strlen(p)-1` will be the last character). Note that the copy functions like `strncpy()` go in the "wrong" direction.

```
;> char *p = "hello";
;> p[0];
(char) 'h'
;  p[strlen(p)-1];
(char) 'o'
;> p+3;
(char*) "lo"
;> char buff[10];
;> strncpy(buff,p+1,3); // like substr(1,3). Right to left!
(char*) "ell"
;> std::copy(p,p+1,buff);  // but copy() goes left to right!
;> buff;
(char*) "hll"
;> *p++;
(char) 'h'
;> *p++;
(char) 'e'
;> *p++;
(char) 'l'
```

The function `strtok()` is useful if you need precise control over how a string is broken up into chunks (that is, tokenized). It is a curious function that has some hidden pitfalls. The first time you call it, you pass it the string; thereafter, you pass it `NULL`. It returns a pointer to the tokens and modifies the string in the process; at the end, it returns `NULL`. Essentially, `strtok()` uses `strpbrk()` to find characters from the delimiters, and it sets the ends of tokens to `\0`. So if you don't want the original string argument of `strtok()` to be modified, you must explicitly make a copy first. Also, the library keeps a pointer to the buffer that is being used, so you can tokenize only one string at a time. This can be a problem with a program that has more than one thread of execution.

```
;> char buff[80], *p;
;> strcpy(buff,"one=two[j]");
(char*) "one=two[j]"
;>  strtok(buff,"=");
(char*) "one"
;>  strtok(NULL,"[");
(char*) "two"
;>  strtok(NULL,"]");
(char*) "j"
;> strcpy(buff,"hello (world)");  // strcpy returns its target
(char *) "hello (world)"
;> p = strtok(buff," ()");
;> while (p != NULL) {
;>   cout << p << endl;
;>   p = strtok(NULL," ()");
;> }
hello
world
;> buff;  // *note that buff has been modified!
(char *) "hello"
```

As with any Application Programming Interface (API), it is wise to wrap eccentric details behind a class facade. The following simple class automatically makes a copy of the buffer to protect it from modification, and it allows you to set sensible default delimiters:

EXAMPLE

```
// tokens.h
class Tokenizer {
private:
  char *m_str;
  char *m_delim;
  bool m_first;
public:
  Tokenizer(char *msg, char *delim = " ")
  {
// make a copy of the buffer
   m_str = new char[strlen(msg)+1];
   strcpy(m_str,msg);
   m_delim = delim;
   m_first = true;
  }
  ~Tokenizer()
   { delete m_str; }

  bool get(std::string &s, char *delim = NULL)
  {
   if (delim == NULL) delim = m_delim;
   char *tok;
   if (m_first) { // first case is special: call strtok() with the string
     tok = strtok(m_str,delim);
     m_first = false;
   } else tok = strtok(NULL,delim);
   if (tok != NULL) s = tok;   // watch out for tok being NULL!
   return tok != NULL;
  }
 };

;> #l tokens.h
;> Tokenizer tt("hello friends");
;> string s;
;> tt.get(s); s;
(bool) true
(string) s = 'hello'
;> tt.get(s); s;
(bool) true
(string) s = 'friends'
;> tt.get(s); s;
(bool) false
(string) s = 'friends'
```

This class still can't be used for multiple streams of tokens, but it is not difficult to write a replacement that doesn't use `strtok()`. This class is particularly useful when tokens are not separated by plain whitespace and when the separators are different for each token. For ordinary token extraction, `istringstream` is more reliable than using `strtok()`.

Miscellaneous Functions: `<cstdlib>`

In the following declarations, `cs` is `const char *`, and `sz` is `size_t` (usually an unsigned long). The character string conversion functions are very useful; `strtol()` in particular not only converts to any arbitrary number system (from base 2 to base 36), but it indicates where an error in a number occurred; if `endp` is not `NULL`, then it is assumed to be a pointer to a C string, which is then filled with any unconverted characters. If the number begins with 0, then the conversion makes the usual C assumptions: leading `0x` means hexadecimal, and leading `0` means octal.

NOTE

I mention octal (that is, base 8) because it is a curious historical relic that can cause trouble; any nonzero integer constant that begins with 0 is interpreted in octal, so, for example, 010 means 8!

```
/* string conversion functions */
double atof(cs);  // converts s to double
int atoi(cs);     // converts s to integer
long atol(cs);    // converts s to long
long strtol(c, char **endp, int base);  // s to long
unsigned long strtoul(c,char **endp,int base); // like strtol
/* pseudo-random number functions */
int  rand();              // result in range 0 to RAND_MAX
void srand(unsigned int); // set a new seed
/* operating system */
int system(cs);    // execute a OS command (like 'dir')
char *getenv(cs);  // get environment string (NULL if none)
/* C memory allocation */
void *malloc(sz);          // allocate sz bytes
void *calloc(sz);          // allocate sz bytes, init. to zero
void *realloc(void *p,sz); // reallocate p to size sz
void  free(void *p);       // deallocates p
```

To get random numbers in the range `0..n`, you use `rand() % n`. Bear in mind that the numbers that `rand()` returns are only pseudorandom numbers; `rand()` produces exactly the same sequence of numbers each time. You should feed `srand()` with something fairly arbitrary such as `clock()` (which is the session time from `<ctime>`) if you need new random sequences each time.

NOTE

I mention C-style memory allocation so you can recognize its use in old code; it is better and easier to use `new` and `delete`. See the section, "Allocatating Memory with `new` and `delete`," in Chapter 5.

Variable-Length Argument Lists: `<cstdarg>`

The <cstdarg> header declares a few functions, beginning with `va_`, which allow you to access arguments in functions that are declared with an unknown number of arguments. For example, the following function `sum()` can add an arbitrary list of integers, provided that the argument list ends with zero. You declare the variable `ap`, which acts as a pointer; it is initialized to the last defined argument (there must be at least one), which is `first` in this case.

Subsequent values are grabbed by using the macro `va_arg()`, which is told the expected type. You must choose a suitable value that will be used to end the argument list, because there is no other way of knowing when the numbers are finished.

```
// in test-stdarg.cpp
int sum(int first,...)
{
  int val = first, result = 0;
  va_list ap;
  va_start(ap,first);
  while (val != 0)
     result += va_arg(ap,int);
  va_end(ap);
  return result;
}
;> sum(10,20,4,0);
(int) 34
```

C++ inherits from C the curious behavior that functions can take an arbitrary number of arguments. This makes the C `printf()` family of functions possible. To allow for an arbitrary number of arugments, the default calling sequence involves pushing the arguments backward (so that the first argument is right on top of the call stack) and making the caller of the function responsible for cleaning up the stack. This is called the `cdecl` calling convention, as opposed to `pascal`, in which arguments are pushed left-to-right and routines clean up their own stack because they always know how many bytes their arguments occupy.

Generally, you should think twice about using variable argument lists in C++. The compiler cannot do any type checking on the extra arguments, and, therefore, you can pass any arbitrary garbage to such functions. It is usually possible to overload operators to achieve the same effect, as with the `iostream` library.

C-Style Input and Output: `<cstdio>`

The key to the C output routines is the `printf()` function, which takes an arbitrary number of arguments. The first argument is a format string, which contains format specifiers. These are matched up one-by-one, with the arguments following the format string. As with C++ stream output, you must explicitly ask for a newline with `\n`, and `printf()`returns the number of characters written out.

```
int printf(const char *format,...);

;> int i = 10, j = 20;
;> printf("%d %d\n",i,j);
10 20
(int) 6
;> double y = 2.3;
;> printf("double %lf value\n",y);
double 2.000000 value
(int) 22
;> double z = sin(y);
;> printf("y=%4.1lf z=%4.2lf\n",y,z);
```

```
y= 2.0 z= 0.92
(int) 15
;> double x = 2.3;
;> printf("%d\n",x);
1717986918
(int) 11
```

A format specifier can contain a width specifier, and in the case of floating-point numbers, it can also contain a precision specifier. The advantage of this style is that it is easy to get exactly the output you want; the disadvantage is (again) that there is no type checking, and you can easily end up with nonsense. In the preceding example, if you try to print a double by using a `%d` specifier, you get a ridiculous number.

All the other members of the `printf()` family work the same way as `printf()`; `fprintf()` writes out to a file opened with `fopen()`, and `sprintf()` writes out to a character buffer. The latter is occasionally still useful as an alternative to a `ostringstream`:

```
;> char buff[80];
;> sprintf(buff,"(%4d,%4d)",k,j);
;> buff;
(char *) " ( 230, 150)"
```

In general, you should stick to C++ streams, which are safer and easier to use than the <cstdio> functions. One somewhat unusual place where the C routines are useful is in embedded programming, where the extra overhead of the `iostream` library may be unacceptable for small devices.

Yet Another Windows Library

Yet Another Windows Library (YAWL), is introduced in Chapter 8, in the section of the same name. I wrote YAWL to make my life as a GUI programmer easier, and it's designed just to do the basics. However, it illustrates the main features of class frameworks, and you will find this knowledge useful if you need to use a real application framework.

This tutorial is organized around the main classes of YAWL; all class names begin with T. `TWin` is the base class for all windows that appear on the screen (although you can make them invisible as well). `TEventWindow` is derived from `TWin` and is the base class for all application windows that need to respond to system events like mouse clicks and keystrokes; this is mostly done by overriding `TEventWindow`'s virtual methods. `TFrameWindow` is derived from `TEventWindow` and is used as the base class for your main application window.

There is also a `TDC`, which encapsulates a Windows Device Context. This is used to access the Windows Graphics Device Interface (GDI) calls, for drawing lines and text, and so forth. A higher-level turtle graphics interface class TG can be used with TGFrameWindow.

Manipulating Windows: (TWin)

TWin encapsulates a window; there are auxilliary classes Rect and Point, which are used to specify screen areas and points.

```
class Rect {
public:
  int left,top,right,bottom;
  Rect() { }
  Rect(int x0, int y0, int x1, int y1)
   { left = x0; top = y0; right = x1;  bottom = y1; }
};
class Point {
public:
  int x,y;
  Point() { }
  Point(int xp, int yp) { x = xp; y = yp; }
};

class TWin {
. . .
public:
// Getting the size and position of the window
  void  get_rect(Rect &rt);           // gets _window_ rectangle
  void  get_client_rect(Rect &rt);    // gets _client_ rectangle
  int   width();
  int   height();
// Converting coordinates
  void  to_screen(Point& pt);
  void  to_client(Point& pt);
 // Resize and move the window
  void  resize(int x0, int y0, int w, int h);
  void  resize(int w, int h);
  void  move(int x0, int y0);
```

The *window rectangle* is the window's position, in screen coordinates. The *client rectangle* is the part of the window that is available for painting (that is, not including scrollbars, menus, caption bars, and so on). The client rectangle is indicated in client coordinates, which are measured from the top-left corner of the client area. Points can be converted between screen and client coordinates by using to_screen() and to_client(). width() and height() refer to the total size, including any nonclient parts such as scrollbars. If you want to find the width of the client area, you check the Rect value that is returned by get_client_rect(). resize() sets the full window size.

These two methods are used to force a repaint.

```
// Causing a repaint
  void  update();
  void  invalidate(Rect *lprt=NULL);
```

Normally, when part of a window is uncovered, that area becomes invalid and causes a paint event to happen, in order to refresh that area of the window (although often people repaint the whole window even if only a small part has changed). `invalidate()` forces a rectangular area to become invalid and thus ready to be repainted; without the `lprt` parameter, the whole client area of the window becomes invalid. Just because a window has invalid areas doesn't mean that it is immediately repainted; paint events wait in a queue for more important things (such as user input) to happen. `update()` is used to force an immediate repaint.

These methods set and get any text associated with the window. The section on user-defined conversions in Chapter 9 gives an example of these methods in use:

```
// Setting and getting the window text
  void  set_text(pchar str);
  pchar get_text(pchar str=obuff,int sz=BUFSIZE);
```

For ordinary top-level windows, the window text is the caption that appears in the top caption bar. For *controls* such as edit boxes, buttons, and labels, the window text is the text displayed in the controls. The `get_text()` method is used to retrieve the text typed in an edit control by a user.

At any point, there is only one active window that is waiting for user input. These functions allow you to access this window, make another window active, change the appearance of a window (for example, make it maximized or even hidden), and so forth:

```
static TWin *get_active_window();
int  get_id();
void set_focus();
void  show(int how = SW_SHOW);
bool  is_visible();
```

If you type `TWin::get_active_window()->set_text("hello")` at the UCW command prompt, the command modifies the actual UCW command window. Likewise, there can be only one window that has the input focus, waiting for keyboard input; you can set this window by using `set_focus()`. You can use `show()` to set the window state; the important states are `SW_HIDE`, `SW_MAXIMIZE`, `SW_MINIMIZE`, `SW_RESTORE`, and `SW_SHOW`. `SW_RESTORE` is used to restore a window to its original state, if it has been minimized, and so forth.

`is_visible()` becomes false after you use `show(SW_HIDE)`.

Managing Events: `TEventWindow`

`TEventWindow` is the base class for any window that tries to intercept Windows events or messages. For the main window you usually use `TFrameWindow`, which is derived from `TEventWindow`. You catch events by overriding the following virtual methods:

```
virtual void size(int cx, int cy);
virtual void on_move();
```

These methods are called when the window is resized or moved, respectively, by user action or directly by `TWin::resize()` and `TWin::move()`.

The following is automatically called to repaint the window when it has become invalid:

```
virtual void paint(TDC& dc);
```

You should try to keep all graphics calls in this method and use the graphics context provided.

The `keydown` event fires when the user presses any key, including noncharacter keys such as Alt and Shift. The following is an example:

```
virtual void keydown(int vkey);
virtual void on_char(int vkey,int repeat);
```

The result of a `keydown()` or a `keyup()` event is an actual character event, which causes `on_char()` to be called. (Note that `repeat` is greater than one if a key has been held down long enough.)

The points shown in the following example are in client coordinates:

```
// mouse messages
 virtual void mouse_down(Point& pt);
 virtual void mouse_move(Point& pt);
 virtual void mouse_up  (Point& pt);
 virtual void right_mouse_down(Point& pt);
 virtual void mouse_double_click(Point& pt);
```

Because `mouse_move()` is called continuously, you should try not to do too much work while processing it.

You can override `command()` if you need to directly process commands (sent from menus, child controls, and so on):

```
virtual bool command(int id);
virtual bool sys_command(int id);
```

These functions must return `true` if you acted on the command, and must return `false` if you ignored it.

The following is called every `m` milliseconds, after a call to `TEventWindow::create_timer(m)`.

```
virtual void timer();
```

You can stop the timer by using `TEventWindow::kill_timer()`.

The following is called whenever the window gains or loses the input focus:

```
virtual void focus(bool yes_no);
```

`query_close()` is called when the user attempts to close a window:

```
virtual bool query_close();
```

This is an opportunity to try to argue with the user and bring up an "are you really sure?" message. If this method returns `false`, the window is not closed. `destroy()` causes the window to close; `TFrameWindow`, for example, redefines `destroy()`, to close the whole application.

Graphics: (`TDC`)

All graphics are created via a *device context* object (in the class `TDC`), which is passed to the `paint` method. You can draw any time you like by defining a `TClientDC` object, which is given an event window pointer.

```
void move_to(int x, int y);
void line_to(int x, int y);
void rectangle(const Rect& rt);
void ellipse(const Rect& rt);
void draw_text(pchar msg);
void text_out(int x, int y, char *buff, int sz = -1);
void set_text_align(int flags, bool update_cp = false);
void get_text_extent(pchar text, int& w, int& h,
                     TFont *font=NULL);
```

These correspond to the common graphics device interface (GDI) calls. `draw_text()` depends on `set_text_align(0,true)`; that is, any `move_to()` or `line_to()` operation modifies `CP` (which stands for "current position"), and the new `CP` value is used to position text. This is currently the default setting.

Setting Color

You can either specify color by using an RGB triplet, where (1.0,1.0,1.0) is pure white, or with a 24-bit hexadecimal color value, such as 0xFF0000. You use the pen to draw lines and outlines of shapes such as rectangles, ellipses, and polygons. You use the brush to fill in shapes. YAWL ties the text and graphics colors together.

```
//*** this changes both the _pen_ and the _text_ color
void set_color(float r, float g, float b);
void set_color(long color);
void set_text_color(long color);
*** ditto, does brush and background color
void set_back_color(float r, float g, float b);
void set_back_color(long color);
```

Setting Text Fonts

You can explicitly construct a `TFont` object and call `TDC::select()`, as in the following example:

```
TFont f1,f2;
f1.create("Arial",24,NORMAL);
f2.create("Courier",12,  BOLD | ITALIC);
...
dc.select(f1);
```

Using Turtle Graphics

The turtle graphics subsystem is an alternative to the traditional GDI-style graphics. I've used turtle graphics in the section "Drawing Trees with Turtle Graphics" in Chapter 6. It brings two new things to the party. First, you can define a floating-point coordinate system that starts at the bottom-left corner. One advantage of this is that the picture can scale automatically with window size.

Second, as well as the usual plotting commands, turtle graphics provides a set of relative graphics commands, organized around the turtle. This mostly invisible beast can be controlled in two ways: by specifying how much it turns and by specifying how much it moves ahead.

You can use `TGFrameWindow`, rather than `TFrameWindow`. The section "Using an Existing Framework" in Chapter 8 has an example of a turtle graphics program that uses `TGFrameWindow`. To use it, override the following methods; they receive points expressed as a pair of floating-point numbers, and the `tg_paint()` method receives its device context as a turtle graphics object.

```
virtual void tg_mouse_down(FPoint& p);
virtual void tg_right_mouse_down(FPoint& p);
virtual void tg_mouse_move(FPoint& p);
virtual void tg_paint(TG& tg);
```

The following are all turtle graphics methods:

```
void scale(double x1, double x2, double y1, double y2);
void penup();
void plot(double x, double y);
void rectangle(double x1, double y1, double x2, double y2);
void ellipse(double x1, double y1, double x2, double y2);
int scalex(double x);
int scaley(double y);
double unscalex(int ix);
double unscaley(int iy);
void fcolor(double r, double g, double b);
void bcolor(double r, double g, double b);
```

You set the domain and range by using `scale()`; the default is (0,100,0,100). The drawing methods are based on the old Hewlett-Packard pen-based API: `plot(x,y)` moves the pen to (x,y); if the pen is down, it draws a line. If the pen is not down, it is moved to the point and then put down. You can explicitly raise the pen by using `penup()`. This example constrasts the two methods of plotting some points. The advantage of using TG over the usual TDC graphics is that the first point is not a special case. The values `x[i]` and `y[i]` in the example would also have to be in client coordinates; in TG, a scale can be set:

EXAMPLE

```
;> // plotting some points using TDC
;> dc.move_to(x[0],y[0]);
;> for(int i = 1; i < n; i++) dc.draw_to(x[i],y[i]);
;> // plotting some points using TG
;> tg.scale(0,n,0,max);
;> tg.penup();
;> for(int i = 0; i < n; i++) tg.plot(i,values[i]);
```

The `scalex()` and `scaley()` routines compute integer client coordinates for the window, and `unscalex()` and `unscaley()` work the other way. You set the foreground and background color with `fcolor()` and `bcolor()`, and `rectangle()` and `ellipse()` work like the `TDC` methods, except that they use scaled coordinates.

The following methods change the turtle angle (in degrees):

```
void turn(double angle);
void left();             // same as turn(-90)
void right();            // same as turn(+90)
```

The following methods move the turtle position, with the pen up and down, respectively:

```
void draw(double len);
void move(double len);
void go_to(double x, double y);
```

The following method shows the turtle (which is initially not visible):

```
void show(bool yes=true);
```

The following method places text at the turtle position:

```
void text(char *msg);
```

C

The C++ Preprocessor

The C/C++ preprocessor is an interesting part of C's history and has been kept in C++ as it developed from C. The preprocessor is not always considered a good thing in C++, and in fact Java, which aimed to correct many of C++'s "mistakes," left out the preprocessor. C programmers do some old-fashioned things with this facility that are considered bad manners in C++, but the preprocessor quietly performs essential things and can occasionally be very useful. In this appendix I will show the full power of the preprocessor, and why some uses are considered bad practice.

continues

Preprocessing Programs

It is a curious fact that C#, which aims to correct the mistakes of Java, has a limited form of the C preprocessor. Originally, when C was first developed, the preprocessor was a separate program, which removed comments, included files, made macro substitutions, did conditional compilation, and then passed the resulting output to the compiler. That is, the preprocessor is the 'front end' of a C++ compiler, which always gets to do any processing before passing source onto the compiler, hence the name **pre**-processor. The GCC compiler included with this book has a standalone preprocessor `cpp`, which you can use to see preprocessor output as the compiler sees it. The following DOS command will redirect this output (which is often very long) into another file (by default it writes to standard output):

```
C:\gcc\test> cpp hello.cpp > hello.pcc
```

Most modern compilers, including UnderC, include the preprocessor as an integral part of their input stage for efficiency reasons, but they still have to do the tasks that are discussed in this appendix.

The `#include` Directive

The `#include` directive inserts a source file into the stream of code that the compiler sees. That source file in turn may contain `#include` directives, but the result is a single stream of source code. This is, incidentally, why traditional C++ compilers can be so slow; your program may have 15 lines, but when all the headers are included, the compiler sees thousands, and sometimes even hundreds of thousands, of lines of code.

The simple rule is that `#include "file"` loads the file from the current directory and `#include<file>` loads the file from the `include` directory, but you can specify multiple `include` paths in some compilers. The idea is to separate application header files from library files that are common to all applications.

Macros

The preprocessor allows you to define macros, which are used rather like functions, except that they directly insert text into the source that the compiler sees.

The `#define` Directive

The `#define` directive changes the meaning of a word, or token, so that it is replaced by substitute text. This token is called a C macro. In this example, I have created three macros, `PI`, `IF`, and `THEN`:

```
#define PI 3.1412
#define IF if(
#define THEN )
#define TWOPI 2*PI
```

After these definitions, you can type `IF x > PI THEN`, and the preprocessor replaces the macros with their substitute text; this is called *macro expansion*. The compiler then actually sees `if (PI > 3.1312)`. If the expanded text itself contains a macro, that will be further expanded. The preprocessor does not know about C++ syntax and will even let you redefine keywords; this is a bad idea because you will confuse any normal C++ programmer completely. A common naming convention is to put any macros in uppercase, which makes it clear when they're being used.

Some standard predefined macros are available in any C++ preprocessor; `__FILE__` expands to the current file, and `__LINE__` to the current line in that file.

Macros can have parameters; there must be no space between the name and the opening parenthesis:

```
#define SQR(x) x*x
```

In this example, `SQR(2)` expands as `2 * 2`. Any macros found in the substitution are themselves expanded, so `SQR(PI)` is expanded to `3.1412 * 3.1412`. But `SQR(1+t)` expands as `1+t * 1+t`, which is wrong without parentheses around `1+t` so the macro must be defined as follows:

```
#define SQR(x) (x)*(x)
```

`SQR()` now behaves as expected. Unfortunately, however, `SQR(f(x))` is replaced by `(f(x))*(f(x))`, which means the function `f(x)` is called twice. It's not necessarily wrong (unless the function has side effects), but it could be very inefficient. And `SQR(i++)` is definitely wrong. I'm showing you these problems so you can appreciate that inline template functions do the job much better than macros, and so you can be thankful that nobody **has** to do C anymore. This also emphasizes that macros are not functions and in fact they do a fairly simple-minded substitution.

If you are ever in doubt about the result of a macro substitution, then I encourage you to use the `cpp` utility as described previously. As long as you don't include any system files, the output will be quite short.

Stringizing and Token Pasting

The stringizing operator (`#`) is found only in macro substitutions, and it basically quotes the parameter that follows it, as in the following example:

```
;> #define OUT(val) cout << #val << " = " << val << endl
;> int i = 22;
;> OUT(i);
i = 22

;> #define S(name)  #name " is a dog"
;> S(fred);
(char*) "fred is a dog"
```

Here `OUT(i)` becomes `cout << "i" << " = " << i << endl`; the magic with the second macro `S()` is that adjacent string literals are automatically concatenated by C++ to build up the larger string; that is, `S(fred)` becomes `"fred" " is a dog"`. C++ does this mostly to support **multiline** strings but also to let you do this kind of trick.

```
 string a_long_string =
 "after several days they found themselves "
 "within sight of the breakers around a barren island";
```

After you are finished with a macro, it is possible to undefine it, by using `#undef`. The redefinition of a macro is not an error, but it is considered bad manners, and you get irritating warnings when you try to do it. Also, to `#undef` a macro makes it clear that the macro is used for a particular limited purpose that is now over. For example, the following kind of macro can save a lot of typing in `switch` statements, especially if there are many cases. It is less error prone than a `case` statement, where people often leave off the `break` statement. Notice in the following example that the macro definition and "undefinition" are put as close to the code as possible:

```
char *message(int id)
{
  char *msg;
   switch(id) {
#define CASE(x)  case x:  msg = #x;  break;
  CASE(NOT_FOUND)
  CASE(UNABLE_TO_REACH)
  CASE(DISMISSED)
#undef CASE
 }
 return msg;
}
```

You can put all these `CASE` lines into a header file, called `errors.h` (that is, everything between the `#define` and the `#undef`.) Then you can define an enumeration and an operator to display the header files, like this:

```
enum Errors {
#define CASE(x)  x ,
#include "symbols.h"
#undef CASE
 END_VAL  // I need this dummy at the end w/out a comma…
};
```

```
...
ostream& operator<< (ostream& os, Errors e) {
   char *msg;
  switch(id) {
#define CASE(x)  case x:  msg = #x;  break;
#include "symbols.h"
#undef CASE
   }
 return  os << msg;
}
```

The token-pasting operator (##) allows you to concatenate tokens. Like the stringizing operator, it applies to parameters. This means that new valid C++ identifiers can be constructed:

```
#define INIT(name) init_##name
```

So `INIT(unit)` will expand as `init_unit`. Combining this with the predefined macro `__LINE__`, you can generate a unique name at each source file with `INIT(__LINE__)`, and you will get `init_156`, `init_175`, and so on, depending where the macro is used.

You can extend the substituted text over several lines by using the backspace continuation character (\). Before templates, C++ would produce generic classes like this:

```
#define POINT(t) struct Point_##t { \
         T x,y;                     \
          T(a,b) : x(a),y(b) {}     \
      };
```

Note that the last line of the substitution does not have a continuation. `POINT(float)` would expand to this:

```
struct Point_float { public: float x,y; float(a,b) : x(a),y(b) {}  };
```

This technique would work for generating `POINT(float)`, but you can imagine how clumsy this technique is for serious classes.

When Not to Use Macros

There is one use of macros that nobody—not even C programmers—approves of anymore: using macros for symbolic constants. It is much better to say `const double PI=3.1412` than `#define PI 3.1412`. The type of the constant in the first case is completely explicit, but in the second case it is implicit (not everyone knows that floating-point constants are `double`). Also, in traditional C++ systems, the compiler doesn't know anything about the preprocessor, so it just sees `3.1412` in the second case. Thus, the debugger has no record of `PI`'s existence either, so it cannot be inspected, and you will not be able to browse for the symbol `PI` either. (UnderC is different from normal compilers like GCC in this respect, but this is because the preprocessor is always available when debugging in interactive mode. That is, if `PI` was defined as a macro, then typing `PI` at the ;> prompt would indeed give its value.)

Another problem with macros in C++ has to do with scope; the preprocessor is completely outside the C++ scope system, and so you never know when a macro is going to (silently) damage your program and generally make you wish it had never been born. This is why people insist on writing macros in uppercase (to make them obvious) and always using parameters with their macros; if the macros always have parameters, they will never be confused with constants, which are also traditionally done in uppercase.

When people see the power of macros (especially if they've come from another language and still feel homesick), they often want to make C++ look like their favorite language. You can make C++ look like BASIC or Pascal, but you will not be impressing the person who has to look after your code, who is expecting C++. For example, the following example can be made to compile and run with any C++ compiler:

```
IF a > b OR n < 5 THEN
    PRINT a;
    FOR I = 1 TO n DO
       PRINT I*b;
    NEXT
ENDIF
```

These macros are all straightforward, except for the `FOR` loop. Just for kicks, I'll show you how that one is done:

```
#define FOR  for(int& _ii_ = (
#define TO   );  _ii_ <=
#define DO  ;  _ii_++) {
#define NEXT }
```

`FOR k = 1 TO n DO` expands as follows:

```
for(int& _ii_= (k = 1) ; _ii_ <= n; _ii_++) {
```

You define a temporary reference variable `_ii_` as an alias for the loop variable `i`. The International Standards Organization (ISO) C++ standard promises that this temporary reference is valid only within the `for` loop, so you can use the trick repeatedly. After the reference is bound to the variable, any action (such as `++`) on the reference acts on the variable.

NOTE

Although making C++ look like BASIC is entertaining, it is wise not to take it seriously. You must learn the language as it is: For example, `||` means "or", `&&` means "and," and it is better to use `{` than to use `BEGIN`. No one will stop you from writing some real BASIC occasionally, but don't mix it up with C++. Programs are public documents, and they must be written in a public language. There is a danger of working in a language that is only readable by one person.

There are cases in which one or two control macros can make the language easier to read and maintain. The following is my favorite:

```
#define FOR(k,n)   for(k = 0;  k < (n); k++)
```

Whenever I see `FOR()` in code, I know that it's just the usual 0 to `n-1` loop. When I see a `for(;;)`,I know something different is going on, like a 1 to `n` loop. `for` loops are not easy on the eye, and many people misread them (for instance, one of the `i`s in a loop could be replaced with a `j`). The macro must go as the following two lines:

```
FOR(k,n):
for(k = 0;  k < (n); k++) xxx.
```

Here is another of my favorite macros:

```
#define FORALL(ii,ic)  for(ii = (ic).begin();  ii != (ic).end();  ++ii)
```

This is a very common control statement when iterating over all elements of a container. Writing things like this is often a matter of taste, but I prefer to type `FORALL(ii,ls) ii->do_something();` rather than `for_each (ii.begin(),ii.end(),operation)` because the latter means I still have to define the operation function.

Generally, you should not try to make macros complicated; the simpler they are, the better. Remember that the debugger can tell you nothing about what's going on inside a macro substitution.

Conditional Compilation

Even if macros go completely out of favor, the C++ preprocessor allows you to conditionally compile code, depending on the environment. For example, usually there are debug and release builds of a program; debug builds contain all the symbolic information for the debugger, and release builds are optimized to be fast and/or small. Sometimes you might want slightly different behavior from the debug version of the program. For example, you might have output statements that are for testing only. You might have `assert()`s, and you don't want to burden the release version with the extra code. In these cases, it is useful to exclude code from the compilation.

The `#ifdef` and `#ifndef` Directives

The `#ifdef` directive directs the preprocessor to include the following code, up to a matching `#else` or `#endif`, if the macro symbol following it is defined:

```
#ifdef _DEBUG
  // trace the progress!
    Cerr << "iteration = " << iter << endl;
 #endif
```

In this case the programmer only wants to see the progress of a loop in the debug version, when the symbol _DEBUG is defined.

The #ifndef directive works similarly to the #ifdef directive, except it includes code if a symbol is not defined. A very important use of this is to prevent a header file from being included more than once. (In fact, this is so common that it's worth writing a small program to generate new header files.) Standard compilers do not allow you to redeclare variables, classes, and so on. The following prevents a header file from being included more than once:

```
// header.h
#ifndef __HEADER_H
#define __HEADER_H
 ...(your code goes here)...
#endif
```

The first time around, the macro __HEADER_H is defined (it does not need a value), and the code is included. The second time around, the macro is defined and the code is not included. In an interactive environment, this can be problematic, so the UnderC load command (#l) automatically undefines any macros defined in the module.

The symbol __UNDERC__ is available only in the UnderC environment, so you can write code that can properly compile under old compilers, which don't have namespaces:

```
#ifdef __UNDERC__
  #include <iostream>
  using namespace std;
#else
  #include <iostream.h>
#endif
```

This ability to configure source to compile properly in all sorts of environments and all sorts of machines was a strong reason that C became so dominant a language. (It is important to note that the compiler does not see any excluded code, so there is no runtime penalty. Conditional compilation has nothing to do with C++ if-else statements.)

Another simple application is to produce an assert() statement. These can be very useful in debugging; if the asserted statement is no longer true, then the assert() will stop the program, giving the program file and line number, together with the failed statement.

```
int getval(int arr[], int n, int idx) {
  assert(idx >= 0 && idx < n);
  return arr[idx];
}
```

A program containing `getval()` would terminate with the following error message when the index `idx` is out of range:

```
assertion failed: test_getval.cpp; idx >= 0 && idx < n
```

The standard `assert()` is available from the `<assert.h>` system header, but the following is a simplified implementation of `assert()`, which shows how `assert()` statements literally expand to nothing when you don't need them:

```
#ifdef _DEBUG
  #define assert(expr)  if (!(expr))  __assert(__FILE__ "; " #expr)
#else
 #define assert(expr)
#endif
```

A simple version of the function `__assert()` puts out the message and terminates the program with `exit()`, and would look like this:

```
void __assert(char *msg)
{
   cerr << "assertion failed: " << msg << endl;
   exit(-1);
}
```

The `#if` Directive

The `#if` directive is a generalization of `#ifdef`, and it allows you to specify a conditional expression. This looks very much like a C/C++ expression, but it is restricted to compile-time constants that involve macros. The special function `defined()` can be used to inquire about a macro. For example,

```
#if defined(_DEBUG) && _TRACE > 2
#message "Now debugging…"
#endif
```

Currently UnderC does not support this full syntax.

Note the `#message` directive; it is occaisonally useful to print out a message when the compiler is preprocessing. This will appear when the programmer builds the code.

D

Compiling C++ Programs and DLLs with GCC and BCC32

UnderC is designed to make beginning C++ programming easier, but in order to master C++, you have to learn the serious tools of the trade. This appendix gives details on how to use two well-known freely available compilers, the GNU C++ compiler (GCC) and the Borland Free C++ compiler (BCC32), which is essentially the engine that powers Borland's C++ Builder. None of the free compilers includes an integrated development environment (IDE), so the *C++ by Example* Website carries a version of Quincy 2000 (`home.mweb.co.za/sd/sdonovan/ccbx.htm`), by Al Stevens, that recognizes either GCC or BCC32 and allows you to set up projects. I've modified this IDE to work with UnderC as well, so you can easily switch between the interpreter and the compilers. Quincy 2000 comes with online help, which I encourage you to consult.

A good IDE is very useful (Visual Studio has become my second home), but it is also useful to know how to compile and link C++ programs from the command line.

Getting Free Compilers

GCC is freely available under terms of the GNU Public License (GPL) and is included on the CD-ROM that accompanies this book. The GPL places no restrictions on what you can do with your own executables. The version that is included on the CD-ROM is version 2.95.2, the Mingw32 edition (which is short for Minimal GNU Win32.) This edition contains all the header files and import libraries for working with both text-based and GUI Windows applications. Unlike with the better-known Cygwin, with Mingw32, programs are dependent only on the standard Microsoft runtime dynamic link library (DLL) `MSVCRT40.DLL` and are freely distributable.

TIP

To receive updates on Mingw32, see the Mingw32 project at `http://sourceforge.net/projects/mingw32`. There is also an excellent mailing list you can consult if you have any trouble getting Mingw32 programs to run properly.

The installation procedure for Mingw32 is straightforward: If your install directory is `c:\GCC`, then you add `c:\GCC\bin` to the path (for example, `set PATH=%PATH%;c:\GCC\bin`). With Windows 9x, you can add this path to `AUTOEXEC.BAT`; remember that this file can be specified for each command prompt shortcut. For Windows NT/2000, you can specify the path in the command prompt properties. After you have added the path, the command `c++` is available at the command prompt.

BCC32 is free for personal use and can also be found on the CD-ROM. As with GCC, if your installation directory is `c:\Borland\BCC55`, then you need to add `c:\Borland\BCC55\bin` to the path. You also have to create two `.cfg` files, which tell the compiler where to find the include and library files. The compiler name at the command prompt is `bcc32`.

```
contents of bcc32.cfg
-I"c:\Borland\Bcc55\include"
-L"c:\Borland\Bcc55\lib"

contents of ilink32.cfg
-L"c:\Borland\Bcc55\lib"
```

Quincy 2000 recognizes the GCC and BCC32 compilers automatically, as long as you give it the right directories. Under Tools, Options, you can click on the Directories tab and type, for example, `c:\gcc` or `c:\borland\bcc55` into the Compiler field. Quincy 2000 then locates the include and library files for you. In the Run tab, you must turn off the Use UCW checkbox if you want to use the installed compiler; this item also appears on the Tools menu.

TIP

It is very useful to have more than one compiler available; when you get mysterious error messages, it's good to have a second opinion. Therefore, you should use both compilers to get a better understanding of where your programs are going wrong.

The Compilation Process: Compile, Link, Go

UnderC is quick at getting programs running, but not very fast at executing them. Once you are happy with your program and need to generate a standalone `.EXE` file, you need to *build* the program using a C++ compiler and linker.

A C++ compiler translates `.cpp` files into object files that contain machine code. These files can't be executed immediately because they contain unresolved references to other functions, some of which are in the other object files of the project but most of which are contained in the libraries. In the case of GCC, the object files end in `.o`, and the library files end in `.a` (for "archive"). For BCC32 and Microsoft C++ (CL), the object files end in `.obj`, and the library files end in `.lib`.

The linker resolves all the function references in the object files, packing them together into an executable program. Not all the code a program needs is necessarily linked statically; much of the runtime libraries sit in `MSVCRT40.DLL`, which is a DLL. (DLLs are often called *shared libraries*.) DLLs are loaded only when the program begins execution. (Windows is mostly a collection of DLLs plus device drivers.) The file that results from the link phase is by default called `a.exe` in GCC, but it is easy to modify the name by using the `-o` command-line option, which is followed by the desired name, as in the following example:

```
C:\gcc\test>c++ hello.cpp
C:\gcc\test>a 10 20
Program is C:\GCC\TEST\A.EXE
arg 0 C:\GCC\TEST\A.EXE
arg 1 10
arg 2 2
C:\gcc\test>c++ -o hello.exe -s hello.cpp
C:\gcc\test>hello
Program is C:\GCC\TEST\HELLO.EXE
```

The resulting executable from the first command is about 167KB, which seems large for a 10-line program, but GCC defaults to outputting debugging information. So using the `-s` command-line option (which *strips* the debug information from the file) brings the file size down to 75KB.

If you really need a small program and can live with some restrictions, you can do two things: You can use C input and output (that is, `printf()`, and so forth) or you can tell the compiler to use the UnderC pocket libraries. You use the `-I` command-line option to specify the UnderC for Windows (UCW) include directories. If the UCW directory is `c:\UCW`, the following command builds a 23KB "Hello, World!" C++ program that is noticeably faster to compile and link than using the default C++ libraries. This shows that there is nothing intrinsically big and bloated about C++ programs; it all depends on the libraries.

```
C:\gcc\test>c++ -o hello.exe -s -I"c:\ucw\include" hello.cpp
```

In Quincy 2000, you can achieve the same effect by selecting the Build tab of the Options dialog, and putting `c:\ucw\include` in the Includes text field.

To build a program that contains a number of files, you can put more than one `.cpp` file on the command line. The following example first shows you the one-line command to build a program composed of two files, and then breaks this down into two separate compilation steps and one separate link step (note the `-c` option, which tells GCC to compile but not link):

```
C:\gcc\test>c++ -o test.exe one.cpp two.cpp
C:\gcc\test>c++ -c one.cpp
C:\gcc\test>c++ -c two.cpp
C:\gcc\test>c++ -o test.exe one.o two.o
```

The trouble with building this program on one line is that each file is compiled separately, whether it needs to be recompiled or not. This gets to be a hassle for anything more than a simple project. Using `-c`, you can compile a C++ file manually, but then you need to recompile anything that might have changed—remember that a file must be recompiled if any header (include) file changes. Creating a project in Quincy 2000 is one way to build a program efficiently; later in this appendix, you will learn how to put together a simple makefile.

Building a YAWL Application

The installation on the CD-ROM comes with two YAWL libraries: `libyawl.a` (GCC) and `yawl.lib` (BCC32). YAWL is used for writing graphical Windows programs and stands for Yet Another Windows Library, and is discussed in Appendix B, "A Short Library Reference." To build a YAWL program with GCC, you need to link with `libyawl.a` and the Mingw32 graphics device interface (GDI) import library:

```
C:\gcc\test>c++ yhello.cpp libyawl.a -lgdi32 -lcomdlg32
```

You need the GDI library to glue YAWL to the underlying Windows application programming interface (API) graphics calls. In the Mingw32 `lib` directory of GCC, you find a large number of files like `lib*.a`, one of which is `libgdi32.a`. The name following the `-l` option in the command line is prepended with `lib`, and then `.a` is appended. For the preceding command-line example to work, `libyawl.a` must be in the current directory. It is most convenient to move it into `c:\gcc\lib` (wherever GCC is located) because you can then access it as follows:

```
C:\gcc\test>c++ yhello.cpp -lyawl -lgdi32 -lcomdlg32
```

This works, but the result has an interesting property. The resulting program has both a window and a text console. If the program is run from a command prompt, an extra console window is not generated. If the program is run from Windows Explorer, the extra console window can be irritating. (Sometimes it can be helpful to use the console for debug output.) So here is the final version of how to build a YAWL program, which you might want to make into a batch file:

```
C:\gcc\test>c++ yhello.cpp -lyawl -lgdi32 -lcomdlg32 -mwindows
```

Linking a DLL into UCW

Here is the UCW header file for the `<cctype>` character classification functions:

```
// UnderC Development Project, 2001
#ifndef __cctype_H
#define __cctype_H
#lib MSVCRT40.DLL
extern "C" {
  int isalpha(int);
  int isalnum(int);
  int isdigit(int);
  int isspace(int);
  int isprint(int);
  char toupper(int);
  char tolower(int);
}
#lib
#endif
```

The `#lib` command is unique to UnderC. It dynamically loads the specified DLL. Thereafter, any function prototype is assumed to be a reference to that DLL, until a `#lib` command appears without a filename.

The extern “C” qualifier is standard C++ and requires some explanation. Normally, C++ function names within libraries are mangled or decorated; that is, the function name has extra information about the signature.

For example, the following is an Microsoft mangled name: `?getline@istream@@QAEAAV1@PACHD@Z`. Name mangling is not done to confuse programmers; rather, it is a consequence of function overloading. Name mangling allows the linker to find exactly the right function from an overloaded set; there is actually more than one function called istream::getline(). Although mangling occurs in a number of ways depending on the compiler, it is not the major obstacle to compatibility between different compilers. More important is the fact that implementors have done things like passing and returning object values in different ways.

`extern "C"` tells the compiler that the function name is not mangled as a C++ name but imported (or exported) as a plain C-style function. In a similar way, to export a C++ function to another language, you use this:

```
extern "C" double sqr(double x)
{ return x*x; }
```

In this way, C++ programs can be linked with other languages such as C, Pascal, and FORTRAN.

You can extract the exported functions from a DLL by using the TDUMP utility that comes with the Borland compiler. For example, you can find the mangled form of `istream::getline()` by dumping the exports of `MSVCRT40.DLL` into a redirected file `out.txt`. `MSVCRT40.DLL` is the Microsoft runtime library that ships with Windows:

```
C:\gcc\test>tdump -ee \windows\system\msvcrt40.dll > out.txt
```

Fortunately, the standard C functions are exported with plain undecorated names, otherwise only Microsoft C++ programs could use them. Any function in this DLL that you need to export must therefore include `extern "C"`.

You can import any Windows API call by linking to the correct DLL. Most window-management functions are in `user32.dll`; note that any function that deals with text comes in ASCII and Unicode versions, which are distinguished by A and W, respectively. (All these names are publicly available through TDUMP.) The `__API` attribute is unique to UCW and is necessary because the calling convention (`__stdcall`) is different from usual C calls. `__stdcall` is also an available attribute, but the name mangling is nonstandard (that is, Microsoft does not follow its own conventions). This example shows how two Windows API functions, `FindWindow()` and `SetWindowText()`, can be imported into an UnderC interactive session. Note the A for ASCII at the end of the function names:

```
;> #lib user32.dll
;> extern "C"
;>  __API int FindWindowA(char *,char *);
;> extern "C"
;>  __API void SetWindowTextA(int,char *);
;> int h;
;> h = FindWindowA(NULL,"UnderC for Windows");
(int) 1332
;> SetWindowTextA(h,"hello dolly");
```

The `FindWindowA()` API call finds the first window with the specified caption and returns an integer value called the *window handle*—in this session, the UCW console window. Using the window handle, you can change the window text, which for top-level windows is the caption. Interactively experimenting with the Windows API is a good way to become familiar with the available resources. Bear in mind that there is practically no overhead involved in using the API because DLLs such as `user32.dll` have already been loaded.

Building an API program with GCC is straightforward because the prototypes are all available in `<windows.h>`. You don't need to remember the `A` because `FindWindow` is a macro defined as `FindWindowA` if you're using ASCII. GCC by default uses the import libraries for `user32.dll` and `kernel32.dll`, which covers most common API calls, except for graphics (in which case you use `-lgdi` as with linking YAWL programs). Here is a standalone version of the preceding UCW session which will compile with GCC (or any Windows compiler):

```
// wtext.cpp
#include <windows.h>
int main()
{
 HWND h = FindWindow(NULL,"UnderC for Windows");
 SetWindowText(h,"hello dolly");
}
c:\gcc\test> c++ -s wtext.cpp
```

This produces a 4KB program, which again proves that C++ isn't always bloated. However, compiling `wtext.cpp` is not fast because of the sheer size of `<windows.h>`. UnderC is such a useful prototyping tool for API programming because it is practically instaneous.

Building a DLL with GCC

A large program can be built using several DLLs, which can be tested independently and even loaded (and unloaded) dynamically. This makes it easy for a team of people to work on the same project. Also, on large projects, the greatest part of the build time is in the linking phase, which is much shorter for DLLs than for static linking, where the linker has to find the functions in the libraries and physically build them into the program file. UnderC is useful in this case because the performance-critical parts of the program can be built as DLLs and then tested and glued together in the interactive environment. So you can have the advantages of both fast code and interactive development.

If you mix compilers, the best bet is to export all functions from a DLL as C-style functions. These functions can be implemented using C++, so there's no need to do everything in C. Here is a very simple DLL source file:

```
// testdll.cpp
#define EXPORT extern "C" __declspec(dllexport)

EXPORT double sqr(double x)
{
 return x*x;
}
```

`extern "C"` forces the name to remain unmangled, and `__declspec(dllexport)` is how a Windows compiler indicates that a function is to be exported. (It's best to make that a macro.) Another way to indicate the exports is to list them in a `.DEF` file, but the method in the example is generally simpler to manage.

The command to build the preceding DLL with GCC is as follows:

```
c:\dlltest> c++ –shared -o dlltest.dll dlltest.cpp
```

This DLL can be immediately loaded into UCW, as follows:

```
;> #lib dlltest.dll
;> extern "C" double sqr(double);
;> sqr(2.0);
(double) 4.0
```

The UnderC command `#unload dlltest.dll` will release the connection with the DLL; it is then possible to rebuild the DLL without getting a sharing violation or having to close the UnderC session. You can then bring the DLL back in with the first two commands above (that is, re-including the prototype as well). It is useful to put any imports into a header file for this reason.

A Simple Makefile

A complex project becomes tricky to build—or to *make*—because each source file can have several dependencies. Say that a project has two source files, `one.cpp` and `two.cpp`, and there is also a header (`one.h`). Both source files are dependent on the header. Here is a makefile to build this project:

```
project.exe : one.o two.o
     c++ -o project.exe one.o two.o
one.o : one.cpp one.h
```

```
     c++ -c one.cpp
two.o : two.cpp one.h
     c++ -c two.cpp
```

A makefile consists of rules that have three parts: a target, a colon followed by the dependent files, and a command that generates the target from the dependent files. The first rule in our example is for `project.exe`, which is dependent on the two object files `one.o` and `two.o`. The next line is the command that builds `project.exe` using the GCC linker.

If `two.cpp` changes, the third rule says that `two.o` is dependent on `two.cpp` , so it is recompiled; the first rule says that `project.exe` is dependent on `two.o`, so the program is relinked.

Here is a more convenient form of the makefile, which puts the list of object files into a variable and defines an implicit rule for building `.o` files from `.cpp` files. A variable `x` in a makefile is accessed with `$(x)`; the special symbol `$<` refers to the dependency of the implicit rule, which will be the `.cpp` file in this case:

```
ofiles = tg-yawl.o twl.o tree.o
yowl.exe : $(ofiles)
     c++ -o yowl.exe $(ofiles) -lgdi32
.cpp.o :
  c++ -c $<
tg-yawl.o : tg-yawl.cpp twl.h
twl.o : twl.cpp twl.h
tree.o : tree.cpp turtle.h
```

The list of object files (which can stretch over several lines for a large project) is now kept in one place, and the implicit rule `.cpp.o` defines what it means to convert a `.cpp` file into an `.h` file.

If you call this file `makefile`, the command `make` at the Windows prompt builds the program, but only if anything has changed. The GNU documentation that comes with the GCC installation has an excellent manual for `make`. One issue I have noticed is that if you have installed the Borland compiler as well, the Borland program of the same name may be first on the path, which will cause confusion since it is more old-fashioned than GNU `make`. In such cases rename `make.exe` in your Borland `bin` directory.

Using Compiler Extensions

Every C++ compiler has a few special features. These features can be helpful, as long as you keep in mind that they are not portable. For example, GCC's `typeof` operator has also (more or less independently) been implemented in UnderC. (It is on Bjarne Stroustrup's list of things he would like to see in the next standard C++ revision, so its future is promising.) The basic idea is that you can use it in declarations, where `type` usually has the following form:

```
;> int i;
;> typeof(i) k,l,m;
```

This might seem like a roundabout way of declaring integer variables, but some entertaining things become possible. Control-statement macros become even more expressive with the use of `typeof`. For example, it is common to use an iterator to access all of the elements in a standard container. It's possible to save typing the loop out by using the following macro:

```
#define FORALL(it,c) \
   for(typeof::iterator it = c.begin(); it != c.end(); ++it)
;> FORALL(ii,ls) s += *ii;  // concatenate all strings in ls
```

This version of `FORALL` unfortunately cannot be implemented by using standard C++, which means it should be avoided in portable code. However, it can be useful in an interactive UnderC setting or if you use GCC exclusively.

Several languages, including C#, have `for each` constructs. Here is how a C++ `FOR_EACH` can be implemented:

```
template <class C, class T>
   struct _ForEach {
     typename C::iterator m_it,m_end;
     T& m_var;

    _ForEach(C& c, T& t) : m_var(t)
     { m_it = c.begin(); m_end = c.end(); }

     bool get() {
       bool res = m_it != m_end;
       if (res) m_var = *m_it;
       return res;
     }

     void next() { ++m_it; }
   };
;> string s;
;> _ForEach<list<string>,string > sri(ls,s);
;> sri.get();
(bool) true
;> s;
(string) 'one';
;> sri.next();
;> sri.get();
```

```
(bool) true
;> s;
(string) 'two';
```

You need a class `_ForEach` that sets a given variable to each element of a container in turn. Such a class, which can be called a *reference iterator*, is easy to define.

This object keeps a reference to the variable, and it uses an iterator to the container. It works for any type that supports an iterator type—not just for the standard containers. It has two type parameters because it's nice to be free to use any compatible type for the loop variable. Given this class, the following is the `FOR_EACH` macro:

```
#define FOR_EACH(v,c) \
  for(_ForEach<typeof,typeof(v)> _fe(c,v); \
      _fe.get();  _fe.next())
;> FOR_EACH(s,ls) cout << s << endl;
one
two
;> int i;
;> FOR_EACH(i,s) cout << i << endl;
116
119
111
```

This example first displays all the strings in a list of strings `ls`, and then dumps out the ASCII values for the characters in `s`.

Curiously, it is possible to implement `FOR_EACH` in standard C++. You need to write a function template that can deduce the type parameters and then create the reference iterator. However, you cannot declare a variable of the specific type, so `ForEach` needs to derive from a base class that defines `get()` and `next()` virtual methods, and this has to be dynamically allocated, so you need to use `auto_ptr()` to make sure this object is disposed of properly. Amazingly, the whole thing works, but it does not break any speed records because of the virtual method calls. The version using `typeof`, however, is practically as fast as an explicit iterator; you will need to watch out for any big objects being copied to the variable.

Index