C++ Programming

Sharam Hekmat
Pragmatix Software Pty. Ltd.
www.pragsoft.com

Contents

Preface

Since its introduction less than a decade ago, C++ has experienced growing acceptance as a practical object-oriented programming language suitable for teaching, research, and commercial software development. The language has also rapidly evolved during this period and acquired a number of new features (e.g., templates and exception handling) which have added to its richness.

This book serves as an introduction to the C++ language. It teaches how to program in C++ and how to properly use its features. It does *not* attempt to teach object-oriented design to any depth, which I believe is best covered in a book in its own right.

In designing this book, I have strived to achieve three goals. First, to produce a *concise* introductory text, free from unnecessary verbosity, so that beginners can develop a good understanding of the language in a short period of time. Second, I have tried to combine a *tutorial* style (based on explanation of concepts through examples) with a *reference* style (based on a flat structure). As a result, each chapter consists of a list of relatively short sections (mostly one or two pages), with no further subdivision. This, I hope, further simplifies the reader's task. Finally, I have consciously avoided trying to present an absolutely complete description of C++. While no important topic has been omitted, descriptions of some of the minor idiosyncrasies have been avoided for the sake of clarity and to avoid overwhelming beginners with too much information. Experience suggests that any small knowledge gaps left as a result, will be easily filled over time through self-discovery.

Intended Audience

This book introduces C++ as an object-oriented programming language. No previous knowledge of C or any other programming language is assumed. Readers who have already been exposed to a high-level programming language (such as C or Pascal) will be able to skip over some of the earlier material in this book.

Although the book is primarily designed for use in undergraduate computer science courses, it will be equally useful to professional programmers and hobbyists who intend to learn the language on their own. The entire book can be easily covered in 10-15 lectures, making it suitable for a one-term or one-semester course. It can also be used as the basis of an intensive 4-5 day industrial training course.

Structure of the Book

The book is divided into 12 chapters. Each chapter has a flat structure, consisting of an unnumbered sequence of sections, most of which are limited to one or two pages. The aim is to present each new topic in a confined space so that it can be quickly grasped. Each chapter ends with a list of exercises. Answers to all of the exercises are provided in an appendix. Readers are encouraged to attempt as many of the exercises as feasible and to compare their solutions against the ones provided.

For the convenience of readers, the sample programs presented in this book (including the solutions to the exercises) and provided in electronic form.

Sharam Hekmat
Melbourne, Australia

1. Preliminaries

This chapter introduces the basic elements of a C++ program. We will use simple examples to show the structure of C++ programs and the way they are compiled. Elementary concepts such as constants, variables, and their storage in memory will also be discussed.

The following is a cursory description of the concept of programming for the benefit of those who are new to the subject.

Programming

A digital computer is a useful tool for solving a great variety of **problems**. A solution to a problem is called an **algorithm**; it describes the sequence of steps to be performed for the problem to be solved. A simple example of a problem and an algorithm for it would be:

Problem: Sort a list of names in ascending lexicographic order.
Algorithm: Call the given list *list1*; create an empty list, *list2*, to hold the sorted list. Repeatedly find the 'smallest' name in *list1*, remove it from *list1*, and make it the next entry of *list2*, until *list1* is empty.

An algorithm is expressed in abstract terms. To be intelligible to a computer, it needs to be expressed in a language understood by it. The only language really understood by a computer is its own **machine language**. Programs expressed in the machine language are said to be **executable**. A program written in any other language needs to be first translated to the machine language before it can be executed.

A machine language is far too cryptic to be suitable for the direct use of programmers. A further abstraction of this language is the **assembly language** which provides mnemonic names for the instructions and a more intelligible notation for the data. An assembly language program is translated to machine language by a translator called an **assembler**.

Even assembly languages are difficult to work with. High-level languages such as C++ provide a much more convenient notation for implementing algorithms. They liberate programmers from having to think in very low-level terms, and help them to focus on the algorithm instead. A program written in a high-level language is translated to assembly language by a translator called a **compiler**. The assembly code produced by the compiler is then assembled to produce an executable program.

A Simple C++ Program

Listing 1.1 shows our first C++ program, which when run, simply outputs the message `Hello World`.

Listing 1.1

```
1  #include <iostream.h>

2  int main (void)
3  {
4      cout << "Hello World\n";
5  }
```

Annotation

1 This line uses the **preprocessor directive** `#include` to include the contents of the header file `iostream.h` in the program. `Iostream.h` is a standard C++ header file and contains definitions for input and output.

2 This line defines a **function** called `main`. A function may have zero or more **parameters**; these always appear after the function name, between a pair of brackets. The word `void` appearing between the brackets indicates that `main` has no parameters. A function may also have a **return type**; this always appears before the function name. The return type for `main` is `int` (i.e., an integer number). All C++ programs must have exactly one `main` function. Program execution always begins from `main`.

3 This brace marks the beginning of the body of `main`.

4 This line is a **statement**. A statement is a computation step which may produce a value. The end of a statement is always marked with a semicolon (`;`). This statement causes the **string** `"Hello World\n"` to be sent to the `cout` **output stream**. A string is any sequence of characters enclosed in double-quotes. The last character in this string (`\n`) is a newline character which is similar to a carriage return on a type writer. A stream is an object which performs input or output. `Cout` is the standard output stream in C++ (standard output usually means your computer monitor screen). The symbol `<<` is an **output operator** which takes an output stream as its left operand and an **expression** as its right operand, and causes the value of the latter to be sent to the former. In this case, the effect is that the string `"Hello World\n"` is sent to `cout`, causing it to be printed on the computer monitor screen.

5 This brace marks the end of the body of `main`. □

Compiling a Simple C++ Program

Dialog 1.1 shows how the program in Listing 1.1 is compiled and run in a typical UNIX environment. User input appears in **bold** and system response in plain. The UNIX command line prompt appears as a dollar symbol ($).

Dialog 1.1

```
1  $ CC hello.cc
2  $ a.out
3  Hello World
4  $
```

Annotation

1 The command for invoking the AT&T C++ translator in a UNIX environment is CC. The argument to this command (hello.cc) is the name of the file which contains the program. As a convention, the file name should end in .c, .C, or .cc. (This ending may be different in other systems.)

2 The result of compilation is an executable file which is by default named a.out. To run the program, we just use a.out as a command.

3 This is the output produced by the program.

4 The return of the system prompt indicates that the program has completed its execution.

The CC command accepts a variety of useful options. An option appears as -name, where name is the name of the option (usually a single letter). Some options take arguments. For example, the output option (-o) allows you to specify a name for the executable file produced by the compiler instead of a.out. Dialog 1.**Erro! Marcador não definido.** illustrates the use of this option by specifying hello as the name of the executable file.

Dialog 1.2

```
1  $ CC hello.cc -o hello
2  $ hello
3  Hello World
4  $
```

Although the actual command may be different depending on the make of the compiler, a similar compilation procedure is used under MS-DOS. Windows-based C++ compilers offer a user-friendly environment where compilation is as simple as choosing a menu command. The naming convention under MS-DOS and Windows is that C++ source file names should end in .cpp. □

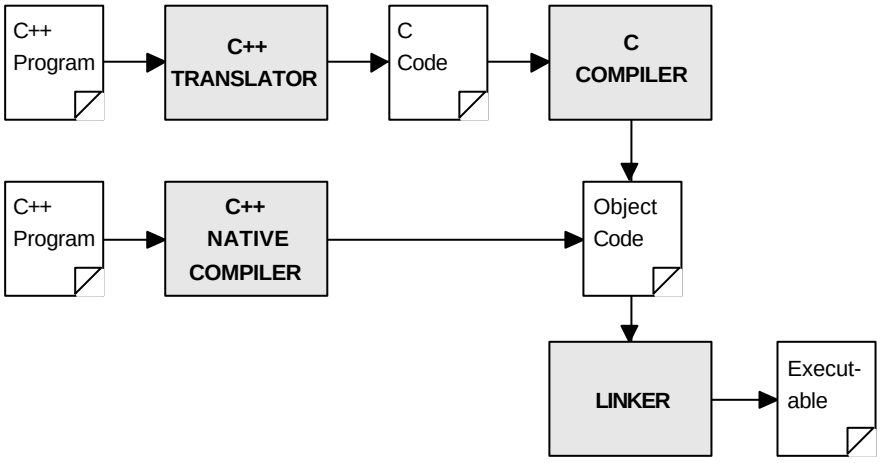
How C++ Compilation Works

Compiling a C++ program involves a number of steps (most of which are transparent to the user):

- First, the C++ **preprocessor** goes over the program text and carries out the instructions specified by the preprocessor directives (e.g., `#include`). The result is a modified program text which no longer contains any directives. (Chapter 12 describes the preprocessor in detail.)
- Then, the C++ **compiler** translates the program code. The compiler may be a true C++ compiler which generates native (assembly or machine) code, or just a translator which translates the code into C. In the latter case, the resulting C code is then passed through a C compiler to produce native object code. In either case, the outcome may be incomplete due to the program referring to library routines which are not defined as a part of the program. For example, Listing 1.1 refers to the `<<` operator which is actually defined in a separate IO library.
- Finally, the **linker** completes the object code by linking it with the object code of any library modules that the program may have referred to. The final result is an executable file.

Figure 1.1 illustrates the above steps for both a C++ translator and a C++ native compiler. In practice all these steps are usually invoked by a single command (e.g., `CC`) and the user will not even see the intermediate files generated.

Figure 1.1 C++ Compilation



□

Variables

A variable is a symbolic name for a memory location in which data can be stored and subsequently recalled. Variables are used for holding data values so that they can be utilized in various computations in a program. All variables have two important attributes:

- A **type** which is established when the variable is defined (e.g., integer, real, character). Once defined, the type of a C++ variable cannot be changed.
- A **value** which can be changed by assigning a new value to the variable. The kind of values a variable can assume depends on its type. For example, an integer variable can only take integer values (e.g., 2, 100, -12).

Listing 1.2 illustrates the uses of some simple variable.

Listing 1.2

```
 1  #include <iostream.h>

 2  int main (void)
 3  {
 4      int     workDays;
 5      float   workHours, payRate, weeklyPay;

 6      workDays = 5;
 7      workHours = 7.5;
 8      payRate = 38.55;
 9      weeklyPay = workDays  * workHours * payRate;
10      cout << "Weekly Pay = ";
11      cout << weeklyPay;
12      cout << '\n';
13  }
```

Annotation

4 This line defines an `int` (integer) variable called `workDays`, which will represent the number of working days in a week. As a general rule, a variable is defined by specifying its type first, followed by the variable name, followed by a semicolon.

5 This line defines three `float` (real) variables which, respectively, represent the work hours per day, the hourly pay rate, and the weekly pay. As illustrated by this line, multiple variables of the same type can be defined at once by separating them with commas.

6 This line is an **assignment** statement. It assigns the value `5` to the variable `workDays`. Therefore, after this statement is executed, `workDays` denotes the value 5.

7 This line assigns the value `7.5` to the variable `workHours`.

8 This line assigns the value `38.55` to the variable `payRate`.

9 This line calculates the weekly pay as the product of `workDays`, `workHours`, and `payRate` (`*` is the multiplication operator). The resulting value is stored in `weeklyPay`.

10-12 These lines output three items in sequence: the string `"Weekly Pay = "`, the value of the variable `weeklyPay`, and a newline character.

When run, the program will produce the following output:

```
Weekly Pay = 1445.625
```

When a variable is defined, its value is **undefined** until it is actually assigned one. For example, `weeklyPay` has an undefined value (i.e., whatever happens to be in the memory location which the variable denotes at the time) until line 9 is executed. The assigning of a value to a variable for the first time is called **initialization**. It is important to ensure that a variable is initialized before it is used in any computation.

It is possible to define a variable and initialize it at the same time. This is considered a good programming practice, because it pre-empts the possibility of using the variable prior to it being initialized. Listing 1.3 is a revised version of Listing 1.2 which uses this technique. For all intents and purposes, the two programs are equivalent.

Listing 1.3

```
 1  #include <iostream.h>

 2  int main (void)
 3  {
 4      int     workDays = 5;
 5      float   workHours = 7.5;
 6      float   payRate = 38.55;
 7      float   weeklyPay = workDays * workHours * payRate;

 8      cout << "Weekly Pay = ";
 9      cout << weeklyPay;
10      cout << '\n';
11  }
```

□

Simple Input/Output

The most common way in which a program communicates with the outside world is through simple, character-oriented Input/Output (IO) operations. C++ provides two useful operators for this purpose: `>>` for input and `<<` for output. We have already seen examples of output using `<<`. Listing 1.4 also illustrates the use of `>>` for input.

Listing 1.4

```
1  #include <iostream.h>

2  int main (void)
3  {
4      int     workDays = 5;
5      float   workHours = 7.5;
6      float   payRate, weeklyPay;

7      cout << "What is the hourly pay rate? ";
8      cin >> payRate;

9      weeklyPay = workDays * workHours * payRate;
10     cout << "Weekly Pay = ";
11     cout << weeklyPay;
12     cout << '\n';
13 }
```

Annotation

7 This line outputs the prompt `What is the hourly pay rate?` to seek user input.

8 This line reads the input value typed by the user and copies it to `payRate`. The input operator `>>` takes an **input stream** as its left operand (`cin` is the standard C++ input stream which corresponds to data entered via the keyboard) and a variable (to which the input data is copied) as its right operand.

9-13 The rest of the program is as before.

When run, the program will produce the following output (user input appears in **bold**):

```
What is the hourly pay rate? 33.55
Weekly Pay = 1258.125
```

Both `<<` and `>>` return their left operand as their result, enabling multiple input or multiple output operations to be combined into one statement. This is illustrated by Listing 1.5 which now allows the input of both the daily work hours and the hourly pay rate.

Listing 1.5

```
 1  #include <iostream.h>

 2  int main (void)
 3  {
 4      int     workDays = 5;
 5      float   workHours, payRate, weeklyPay;

 6      cout << "What are the work hours and the hourly pay rate? ";
 7      cin >> workHours >> payRate;

 8      weeklyPay = workDays * workHours * payRate;
 9      cout << "Weekly Pay = " << weeklyPay << '\n';
10  }
```

Annotation

7 This line reads two input values typed by the user and copies them to `workHours` and `payRate`, respectively. The two values should be separated by white space (i.e., one or more space or tab characters). This statement is equivalent to:

```
(cin >> workHours) >> payRate;
```

Because the result of `>>` is its left operand, `(cin >> workHours)` evaluates to `cin` which is then used as the left operand of the next `>>` operator.

9 This line is the result of combining lines 10-12 from Listing 1.4. It outputs `"Weekly Pay = "`, followed by the value of `weeklyPay`, followed by a newline character. This statement is equivalent to:

```
((cout << "Weekly Pay = ") << weeklyPay) << '\n';
```

Because the result of `<<` is its left operand, `(cout << "Weekly Pay = ")` evaluates to `cout` which is then used as the left operand of the next `<<` operator, etc.

When run, the program will produce the following output:

```
What are the work hours and the hourly pay rate? 7.5 33.55
Weekly Pay = 1258.125
```

□

Comments

A comment is a piece of descriptive text which explains some aspect of a program. Program comments are totally ignored by the compiler and are only intended for human readers. C++ provides two types of comment delimiters:

- Anything after `//` (until the end of the line on which it appears) is considered a comment.
- Anything enclosed by the pair `/*` and `*/` is considered a comment.

Listing 1.6 illustrates the use of both forms.

Listing 1.6

```
 1  #include <iostream.h>

 2  /* This program calculates the weekly gross pay for a worker,
 3     based on the total number of hours worked and the hourly pay
 4     rate. */

 5  int main (void)
 6  {
 7      int     workDays = 5;       // Number of work days per week
 8      float   workHours = 7.5;        // Number of work hours per day
 9      float   payRate = 33.50;        // Hourly pay rate
10      float   weeklyPay;          // Gross weekly pay

11      weeklyPay = workDays * workHours * payRate;
12      cout << "Weekly Pay = " << weeklyPay << '\n';
13  }
```

Comments should be used to enhance (not to hinder) the readability of a program. The following two points, in particular, should be noted:

- A comment should be easier to read and understand than the code which it tries to explain. A confusing or unnecessarily-complex comment is worse than no comment at all.
- Over-use of comments can lead to even less readability. A program which contains so much comment that you can hardly see the code can by no means be considered readable.
- Use of descriptive names for variables and other entities in a program, and proper indentation of the code can reduce the need for using comments.

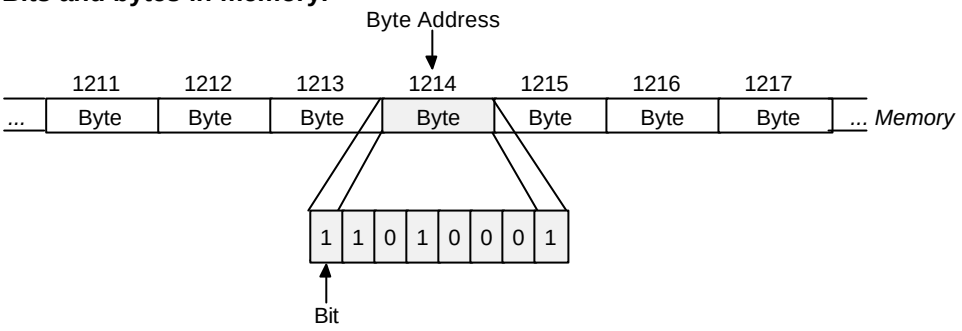
The best guideline for how to use comments is to simply apply common sense.

□

Memory

A computer provides a Random Access Memory (RAM) for storing executable program code as well as the data the program manipulates. This memory can be thought of as a contiguous sequence of **bits**, each of which is capable of storing a **binary digit** (0 or 1). Typically, the memory is also divided into groups of 8 consecutive bits (called **bytes**). The bytes are sequentially addressed. Therefore each byte can be uniquely identified by its **address** (see Figure 1.2).

Figure 1.2 Bits and bytes in memory.

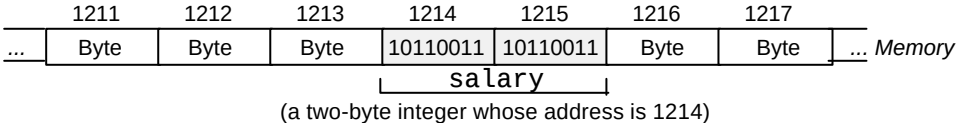


The C++ compiler generates executable code which maps data entities to memory locations. For example, the variable definition

```
int salary = 65000;
```

causes the compiler to allocate a few bytes to represent `salary`. The exact number of bytes allocated and the method used for the binary representation of the integer depends on the specific C++ implementation, but let us say two bytes encoded as a 2's complement integer. The compiler uses the *address* of the first byte at which `salary` is allocated to refer to it. The above assignment causes the value 65000 to be stored as a 2's complement integer in the two bytes allocated (see Figure 1.3).

Figure 1.3 Representation of an integer in memory.



(a two-byte integer whose address is 1214)

While the exact binary representation of a data item is rarely of interest to a programmer, the general organization of memory and use of addresses for referring to data items (as we will see later) is very important.

□

Integer Numbers

An **integer variable** may be defined to be of type `short`, `int`, or `long`. The only difference is that an `int` uses more or at least the same number of bytes as a `short`, and a `long` uses more or at least the same number of bytes as an `int`. For example, on the author's PC, a `short` uses 2 bytes, an `int` also 2 bytes, and a `long` 4 bytes.

```
short   age = 20;
int     salary = 65000;
long    price = 4500000;
```

By default, an integer variable is assumed to be signed (i.e., have a signed representation so that it can assume positive as well as negative values). However, an integer can be defined to be unsigned by using the keyword `unsigned` in its definition. The keyword `signed` is also allowed but is redundant.

```
unsigned short  age = 20;
unsigned int        salary = 65000;
unsigned long   price = 4500000;
```

A **literal integer** (e.g., `1984`) is always assumed to be of type `int`, unless it has an `L` or `l` suffix, in which case it is treated as a `long`. Also, a literal integer can be specified to be unsigned using the suffix `U` or `u`. For example:

```
1984L   1984l   1984U   1984u   1984LU  1984ul
```

Literal integers can be expressed in decimal, octal, and hexadecimal notations. The decimal notation is the one we have been using so far. An integer is taken to be octal if it is preceded by a zero (`0`), and hexadecimal if it is preceded by a `0x` or `0X`. For example:

```
92      // decimal
0134    // equivalent octal
0x5C    // equivalent hexadecimal
```

Octal numbers use the base 8, and can therefore only use the digits `0-7`. Hexadecimal numbers use the base 16, and therefore use the letter `A-F` (or `a-f`) to represent, respectively, `10-15`. Octal and hexadecimal numbers are calculated as follows:

`0134` $= 1 \times 8^2 + 3 \times 8^1 + 4 \times 8^0 = 64 + 24 + 4 = 92$
`0x5C` $= 5 \times 16^1 + 12 \times 16^0 = 80 + 12 = 92$

□

Real Numbers

A **real variable** may be defined to be of type `float` or `double`. The latter uses more bytes and therefore offers a greater range and accuracy for representing real numbers. For example, on the author's PC, a `float` uses 4 and a `double` uses 8 bytes.

```
float   interestRate = 0.06;
double  pi = 3.141592654;
```

A **literal real** (e.g., `0.06`) is always assumed to be of type `double`, unless it has an `F` or `f` suffix, in which case it is treated as a `float`, or an `L` or `l` suffix, in which case it is treated as a `long double`. The latter uses more bytes than a `double` for better accuracy (e.g., 10 bytes on the author's PC). For example:

```
0.06F   0.06f   3.141592654L        3.141592654l
```

In addition to the decimal notation used so far, literal reals may also be expressed in *scientific* notation. For example, 0.002164 may be written in the scientific notation as:

```
2.164E-3        or      2.164e-3
```

The letter `E` (or `e`) stands for *exponent*. The scientific notation is interpreted as follows:

`2.164E-3` $= 2.164 \times 10^{-3}$

□

Characters

A **character variable** is defined to be of type `char`. A character variable occupies a single byte which contains the *code* for the character. This code is a numeric value and depends on the *character coding system* being used (i.e., is machine-dependent). The most common system is ASCII (American Standard Code for Information Interchange). For example, the character *A* has the ASCII code 65, and the character *a* has the ASCII code 97.

```
char    ch = 'A';
```

Like integers, a character variable may be specified to be signed or unsigned. By the default (on most systems) `char` means `signed char`. However, on some systems it may mean `unsigned char`. A signed character variable can hold numeric values in the range -128 through 127. An unsigned character variable can hold numeric values in the range 0 through 255. As a result, both are often used to represent small integers in programs (and can be assigned numeric values like integers):

```
signed char     offset = -88;
unsigned char   row = 2, column = 26;
```

A **literal character** is written by enclosing the character between a pair of single quotes (e.g., `'A'`). Nonprintable characters are represented using escape sequences. For example:

```
'\n'    // new line
'\r'    // carriage return
'\t'    // horizontal tab
'\v'    // vertical tab
'\b'    // backspace
'\f'    // formfeed
```

Single and double quotes and the backslash character can also use the escape notation:

```
'\''    // single quote (')
'\"'    // double quote (")
'\\'    // backslash (\)
```

Literal characters may also be specified using their numeric code value. The general escape sequence `\ooo` (i.e., a backslash followed by up to three octal digits) is used for this purpose. For example (assuming ASCII):

```
'\12'   // newline (decimal code = 10)
'\11'   // horizontal tab (decimal code = 9)
'\101'  // 'A' (decimal code = 65)
'\0'    // null (decimal code = 0)
```

□

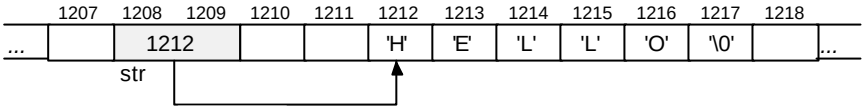
Strings

A string is a consecutive sequence (i.e., *array*) of characters which are terminated by a null character. A **string variable** is defined to be of type `char*` (i.e., a *pointer* to character). A pointer is simply the address of a memory location. (Pointers will be discussed in Chapter 5). A string variable, therefore, simply contains the address of where the first character of a string appears. For example, consider the definition:

```
char    *str = "HELLO";
```

Figure 1.4 illustrates how the string variable `str` and the string `"HELLO"` might appear in memory.

Figure 1.4 A string and a string variable in memory.



A **literal string** is written by enclosing its characters between a pair of double quotes (e.g., `"HELLO"`). The compiler always appends a null character to a literal string to mark its end. The characters of a string may be specified using any of the notations for specifying literal characters. For example:

```
"Name\tAddress\tTelephone"      // tab-separated words
"ASCII character 65: \101"      // 'A' specified as '101'
```

A long string may extend beyond a single line, in which case each of the preceding lines should be terminated by a backslash. For example:

```
"Example to show \
the use of backslash for \
writing a long string"
```

The backslash in this context means that the rest of the string is continued on the next line. The above string is equivalent to the single line string:

```
"Example to show the use of backslash for writing a long string"
```

A common programming error results from confusing a single-character string (e.g., `"A"`) with a single character (e.g., `'A'`). These two are *not* equivalent. The former consists of two bytes (the character `'A'` followed by the character `'\0'`), whereas the latter consists of a single byte.

The shortest possible string is the null string (`""`) which simply consists of the null character. □

Names

Programming languages use names to refer to the various entities that make up a program. We have already seen examples of an important category of such names (i.e., variable names). Other categories include: function names, type names, and macro names, which will be described later in this book.

Names are a programming convenience, which allow the programmer to organize what would otherwise be quantities of plain data into a meaningful and human-readable collection. As a result, no trace of a name is left in the final executable code generated by a compiler. For example, a `temperature` variable eventually becomes a few bytes of memory which is referred to by the executable code by its address, not its name.

C++ imposes the following rules for creating valid names (also called **identifiers**). A name should consist of one or more characters, each of which may be a letter (i.e., 'A'-'Z' and 'a'-'z'), a digit (i.e., '0'-'9'), or an underscore character ('_'), except that the first character may not be a digit. Upper and lower case letters are distinct. For example:

```
salary      // valid identifier
salary2     // valid identifier
2salary     // invalid identifier (begins with a digit)
_salary     // valid identifier
Salary      // valid but distinct from salary
```

C++ imposes no limit on the number of characters in an identifier. However, most implementation do. But the limit is usually so large that it should not cause a concern (e.g., 255 characters).

Certain words are reserved by C++ for specific purposes and may not be used as identifiers. These are called **reserved words** or **keywords** and are summarized in Table 1.1:

Table 1.1 C++ keywords.

`asm`	`continue`	`float`	`new`	`signed`	`try`
`auto`	`default`	`for`	`operator`	`sizeof`	`typedef`
`break`	`delete`	`friend`	`private`	`static`	`union`
`case`	`do`	`goto`	`protected`	`struct`	`unsigned`
`catch`	`double`	`if`	`public`	`switch`	`virtual`
`char`	`else`	`inline`	`register`	`template`	`void`
`class`	`enum`	`int`	`return`	`this`	`volatile`
`const`	`extern`	`long`	`short`	`throw`	`while`

□

Exercises

1.1 Write a program which inputs a temperature reading expressed in Fahrenheit and outputs its equivalent in Celsius, using the formula:

$$^{\circ}C = \frac{5}{9}(^{\circ}F - 32)$$

Compile and run the program. Its behavior should resemble this:

```
Temperature in Fahrenheit: 41
41 degrees Fahrenheit = 5 degrees Celsius
```

1.2 Which of the following represent valid variable definitions?

```
int n = -100;
unsigned int i = -100;
signed int = 2.9;
long m = 2, p = 4;
int 2k;
double x = 2 * m;
float y = y * 2;
unsigned double z = 0.0;
double d = 0.67F;
float f = 0.52L;
signed char = -1786;
char c = '$' + 2;
sign char h = '\111';
char *name = "Peter Pan";
unsigned char *num = "276811";
```

1.3 Which of the following represent valid identifiers?

```
identifier
seven_11
_unique_
gross-income
gross$income
2by2
default
average_weight_of_a_large_pizza
variable
object.oriented
```

1.4 Define variables to represent the following entities:

- Age of a person.
- Income of an employee.
- Number of words in a dictionary.
- A letter of the alphabet.
- A greeting message. □

2. Expressions

This chapter introduces the built-in C++ operators for composing expressions. An expression is any computation which yields a value.

When discussing expressions, we often use the term **evaluation**. For example, we say that an expression evaluates to a certain value. Usually the final value is the only reason for evaluating the expression. However, in some cases, the expression may also produce **side-effects**. These are permanent changes in the program state. In this sense, C++ expressions are different from mathematical expressions.

C++ provides operators for composing arithmetic, relational, logical, bitwise, and conditional expressions. It also provides operators which produce useful side-effects, such as assignment, increment, and decrement. We will look at each category of operators in turn. We will also discuss the precedence rules which govern the order of operator evaluation in a multi-operator expression.

Arithmetic Operators

C++ provides five basic arithmetic operators. These are summarized in Table 2.1.

Table 2.1 Arithmetic operators.

Operator	Name	Example
+	Addition	`12 + 4.9  // gives 16.9`
-	Subtraction	`3.98 - 4  // gives -0.02`
*	Multiplication	`2 * 3.4  // gives 6.8`
/	Division	`9 / 2.0  // gives 4.5`
%	Remainder	`13 % 3  // gives 1`

Except for remainder (`%`) all other arithmetic operators can accept a mix of integer and real operands. Generally, if both operands are integers then the result will be an integer. However, if one or both of the operands are reals then the result will be a real (or `double` to be exact).

When both operands of the division operator (`/`) are integers then the division is performed as an **integer division** and not the normal division we are used to. Integer division always results in an integer outcome (i.e., the result is always rounded down). For example:

```
9 / 2       // gives 4, not 4.5!
-9 / 2      // gives -5, not -4!
```

Unintended integer divisions are a common source of programming errors. To obtain a real division when both operands are integers, you should cast one of the operands to be real:

```
int     cost = 100;
int     volume = 80;
double  unitPrice = cost / (double) volume;      // gives 1.25
```

The remainder operator (`%`) expects integers for both of its operands. It returns the remainder of integer-dividing the operands. For example `13%3` is calculated by integer dividing 13 by 3 to give an outcome of 4 and a remainder of 1; the result is therefore 1.

It is possible for the outcome of an arithmetic operation to be too large for storing in a designated variable. This situation is called an **overflow**. The outcome of an overflow is machine-dependent and therefore undefined. For example:

```
unsigned char   k = 10 * 92;        // overflow: 920 > 255
```

It is illegal to divide a number by zero. This results in a run-time *division-by-zero* failure which typically causes the program to terminate. □

Relational Operators

C++ provides six relational operators for comparing numeric quantities. These are summarized in Table 2.2. Relational operators evaluate to 1 (representing the *true* outcome) or 0 (representing the *false* outcome).

Table 2.2 Relational operators.

Operator	Name	Example
==	Equality	`5 == 5  // gives 1`
!=	Inequality	`5 != 5  // gives 0`
<	Less Than	`5 < 5.5  // gives 1`
<=	Less Than or Equal	`5 <= 5  // gives 1`
>	Greater Than	`5 > 5.5  // gives 0`
>=	Greater Than or Equal	`6.3 >= 5  // gives 1`

Note that the <= and >= operators are only supported in the form shown. In particular, =< and => are both invalid and do not mean anything.

The operands of a relational operator must evaluate to a number. Characters are valid operands since they are represented by numeric values. For example (assuming ASCII coding):

```
'A' < 'F'       // gives 1 (is like 65 < 70)
```

The relational operators should not be used for comparing strings, because this will result in the string *addresses* being compared, not the string contents. For example, the expression

```
"HELLO" < "BYE"
```

causes the address of `"HELLO"` to be compared to the address of `"BYE"`. As these addresses are determined by the compiler (in a machine-dependent manner), the outcome may be 0 or may be 1, and is therefore undefined.

C++ provides library functions (e.g., `strcmp`) for the lexicographic comparison of string. These will be described later in the book.

□

Logical Operators

C++ provides three logical operators for combining logical expression. These are summarized in Table 2.3. Like the relational operators, logical operators evaluate to 1 or 0.

Table 2.3 Logical operators.

Operator	Name	Example	
!	Logical Negation	`!(5 == 5)`	`// gives 0`
&&	Logical And	`5 < 6 && 6 < 6`	`// gives 1`
||	Logical Or	`5 < 6 || 6 < 5`	`// gives 1`

Logical *negation* is a unary operator, which negates the logical value of its single operand. If its operand is nonzero it produce 0, and if it is 0 it produces 1.

Logical *and* produces 0 if one or both of its operands evaluate to 0. Otherwise, it produces 1. Logical *or* produces 0 if both of its operands evaluate to 0. Otherwise, it produces 1.

Note that here we talk of zero and nonzero operands (not zero and 1). In general, any nonzero value can be used to represent the logical *true*, whereas only zero represents the logical *false*. The following are, therefore, all valid logical expressions:

```
!20                 // gives 0
10 && 5             // gives 1
10 || 5.5           // gives 1
10 && 0             // gives 0
```

C++ does not have a built-in boolean type. It is customary to use the type `int` for this purpose instead. For example:

```
int sorted = 0;     // false
int balanced = 1;   // true
```

□

Bitwise Operators

C++ provides six bitwise operators for manipulating the individual bits in an integer quantity. These are summarized in Table 2.4.

Table 2.4 Bitwise operators.

Operator	Name	Example
~	Bitwise Negation	`~'\011'  // gives '\366'`
&	Bitwise And	`'\011' & '\027'  // gives '\001'`
|	Bitwise Or	`'\011' | '\027'  // gives '\037'`
^	Bitwise Exclusive Or	`'\011' ^ '\027'  // gives '\036'`
<<	Bitwise Left Shift	`'\011' << 2  // gives '\044'`
>>	Bitwise Right Shift	`'\011' >> 2  // gives '\002'`

Bitwise operators expect their operands to be integer quantities and treat them as bit sequences. Bitwise *negation* is a unary operator which reverses the bits in its operands. Bitwise *and* compares the corresponding bits of its operands and produces a 1 when both bits are 1, and 0 otherwise. Bitwise *or* compares the corresponding bits of its operands and produces a 0 when both bits are 0, and 1 otherwise. Bitwise *exclusive or* compares the corresponding bits of its operands and produces a 0 when both bits are 1 or both bits are 0, and 1 otherwise.

Bitwise *left shift* operator and bitwise *right shift* operator both take a bit sequence as their left operand and a positive integer quantity *n* as their right operand. The former produces a bit sequence equal to the left operand but which has been shifted *n* bit positions to the left. The latter produces a bit sequence equal to the left operand but which has been shifted *n* bit positions to the right. Vacated bits at either end are set to 0.

Table 2.5 illustrates bit sequences for the sample operands and results in Table 2.4. To avoid worrying about the sign bit (which is machine dependent), it is common to declare a bit sequence as an unsigned quantity:

```
unsigned char x = '\011';
unsigned char y = '\027';
```

Table 2.5 How the bits are calculated.

Example	Octal Value	Bit Sequence							
x	011	0	0	0	0	1	0	0	1
y	027	0	0	0	1	0	1	1	1
~x	366	1	1	1	1	0	1	1	0
x & y	001	0	0	0	0	0	0	0	1
x | y	037	0	0	0	1	1	1	1	1
x ^ y	036	0	0	0	1	1	1	1	0
x << 2	044	0	0	1	0	0	1	0	0
x >> 2	002	0	0	0	0	0	0	1	0

□

Increment/Decrement Operators

The *auto increment* (++) and *auto decrement* (--) operators provide a convenient way of, respectively, adding and subtracting 1 from a numeric variable. These are summarized in Table 2.6. The examples assume the following variable definition:

```
int k = 5;
```

Table 2.6 Increment and decrement operators.

Operator	Name	Example	
++	Auto Increment (prefix)	`++k + 10`	`// gives 16`
++	Auto Increment (postfix)	`k++ + 10`	`// gives 15`
--	Auto Decrement (prefix)	`--k + 10`	`// gives 14`
--	Auto Decrement (postfix)	`k-- + 10`	`// gives 15`

Both operators can be used in prefix and postfix form. The difference is significant. When used in prefix form, the operator is first applied and the outcome is then used in the expression. When used in the postfix form, the expression is evaluated first and then the operator applied.

Both operators may be applied to integer as well as real variables, although in practice real variables are rarely useful in this form.

□

Assignment Operator

The assignment operator is used for storing a value at some memory location (typically denoted by a variable). Its left operand should be an lvalue, and its right operand may be an arbitrary expression. The latter is evaluated and the outcome is stored in the location denoted by the lvalue.

An **lvalue** (standing for **l**eft **value**) is anything that denotes a memory location in which a value may be stored. The only kind of lvalue we have seen so far in this book is a variable. Other kinds of lvalues (based on pointers and references) will be described later in this book.

The assignment operator has a number of variants, obtained by combining it with the arithmetic and bitwise operators. These are summarized in Table 2.7. The examples assume that `n` is an integer variable.

Table 2.7 Assignment operators.

Operator	Example	Equivalent To
`=`	`n = 25`	
`+=`	`n += 25`	`n = n + 25`
`-=`	`n -= 25`	`n = n - 25`
`*=`	`n *= 25`	`n = n * 25`
`/=`	`n /= 25`	`n = n / 25`
`%=`	`n %= 25`	`n = n % 25`
`&=`	`n &= 0xF2F2`	`n = n & 0xF2F2`
`|=`	`n |= 0xF2F2`	`n = n | 0xF2F2`
`^=`	`n ^= 0xF2F2`	`n = n ^ 0xF2F2`
`<<=`	`n <<= 4`	`n = n << 4`
`>>=`	`n >>= 4`	`n = n >> 4`

An assignment operation is itself an expression whose value is the value stored in its left operand. An assignment operation can therefore be used as the right operand of another assignment operation. Any number of assignments can be concatenated in this fashion to form one expression. For example:

```
int m, n, p;
m = n = p = 100;         // means: n = (m = (p = 100));
m = (n = p = 100) + 2;   // means: m = (n = (p = 100)) + 2;
```

This is equally applicable to other forms of assignment. For example:

```
m = 100;
m += n = p = 10;         // means: m = m + (n = p = 10);
```

□

Conditional Operator

The conditional operator takes three operands. It has the general form:

operand1 `?` *operand2* `:` *operand3*

First *operand1* is evaluated, which is treated as a logical condition. If the result is nonzero then *operand2* is evaluated and its value is the final result. Otherwise, *operand3* is evaluated and its value is the final result. For example:

```
int m = 1, n = 2;
int min = (m < n ? m : n);      // min receives 1
```

Note that of the second and the third operands of the conditional operator only one is evaluated. This may be significant when one or both contain side-effects (i.e., their evaluation causes a change to the value of a variable). For example, in

```
int min = (m < n ? m++ : n++);
```

`m` is incremented because `m++` is evaluated but `n` is not incremented because `n++` is not evaluated.

Because a conditional operation is itself an expression, it may be used as an operand of another conditional operation, that is, conditional expressions may be nested. For example:

```
int m = 1, n = 2, p =3;
int min = (m < n ? (m < p ? m : p)
                 : (n < p ? n : p));
```

□

Comma Operator

Multiple expressions can be combined into one expression using the comma operator. The comma operator takes two operands. It first evaluates the left operand and then the right operand, and returns the value of the latter as the final outcome. For example:

```
int m, n, min;
int mCount = 0, nCount = 0;
//...
min = (m < n ? mCount++, m : nCount++, n);
```

Here when `m` is less than `n`, `mCount++` is evaluated and the value of `m` is stored in `min`. Otherwise, `nCount++` is evaluated and the value of `n` is stored in `min`.

□

The sizeof Operator

C++ provides a useful operator, `sizeof`, for calculating the size of any data item or type. It takes a single operand which may be a type name (e.g., `int`) or an expression (e.g., 100) and returns the size of the specified entity in bytes. The outcome is totally machine-dependent. Listing 2.1 illustrates the use of sizeof on the built-in types we have encountered so far.

Listing 2.1

```
 1  #include <iostream.h>

 2  int main (void)
 3  {
 4      cout << "char   size = " << sizeof(char) << " bytes\n";
 5      cout << "char*  size = " << sizeof(char*) << " bytes\n";
 6      cout << "short  size = " << sizeof(short) << " bytes\n";
 7      cout << "int    size = " << sizeof(int) << " bytes\n";
 8      cout << "long   size = " << sizeof(long) << " bytes\n";
 9      cout << "float  size = " << sizeof(float) << " bytes\n";
10      cout << "double size = " << sizeof(double) << " bytes\n";

11      cout << "1.55   size = " << sizeof(1.55) << " bytes\n";
12      cout << "1.55L  size = " << sizeof(1.55L) << " bytes\n";
13      cout << "HELLO  size = " << sizeof("HELLO") << " bytes\n";
14  }
```

When run, the program will produce the following output (on the author's PC):

```
char   size = 1 bytes
char*  size = 2 bytes
short  size = 2 bytes
int    size = 2 bytes
long   size = 4 bytes
float  size = 4 bytes
double size = 8 bytes
1.55   size = 8 bytes
1.55L  size = 10 bytes
HELLO  size = 6 bytes
```

□

Operator Precedence

The order in which operators are evaluated in an expression is significant and is determined by precedence rules. These rules divide the C++ operators into a number of precedence levels (see Table 2.8). Operators in higher levels take precedence over operators in lower levels.

Table 2.8 Operator precedence levels.

Level	Operator						Kind	Order
Highest	`::`						Unary	Both
	`()`	`[]`	`->`	`.`			Binary	Left to Right
	`+` `-`	`++` `--`	`!` `~`	`*` `&`	`new` `delete`	`sizeof` `()`	Unary	Right to Left
	`->*`	`.*`					Binary	Left to Right
	`*`	`/`	`%`				Binary	Left to Right
	`+`	`-`					Binary	Left to Right
	`<<`	`>>`					Binary	Left to Right
	`<`	`<=`	`>`	`>=`			Binary	Left to Right
	`==`	`!=`					Binary	Left to Right
	`&`						Binary	Left to Right
	`^`						Binary	Left to Right
	`|`						Binary	Left to Right
	`&&`						Binary	Left to Right
	`||`						Binary	Left to Right
	`? :`						Ternary	Left to Right
	`=`	`+=` `-=`	`*=` `/=`	`^=` `%=`	`&=` `|=`	`<<=` `>>=`	Binary	Right to Left
Lowest	`,`						Binary	Left to Right

For example, in

```
a == b + c * d
```

`c * d` is evaluated first because `*` has a higher precedence than `+` and `==`. The result is then added to `b` because `+` has a higher precedence than `==`, and then `==` is evaluated. Precedence rules can be overridden using brackets. For example, rewriting the above expression as

```
a == (b + c) * d
```

causes `+` to be evaluated before `*`.

Operators with the same precedence level are evaluated in the order specified by the last column of Table 2.8. For example, in

```
a = b += c
```

the evaluation order is right to left, so first `b += c` is evaluated, followed by `a = b`. □

Simple Type Conversion

A value in any of the built-in types we have see so far can be converted (*type-cast*) to any of the other types. For example:

```
(int) 3.14      // converts 3.14 to an int to give 3
(long) 3.14     // converts 3.14 to a long to give 3L
(double) 2      // converts 2 to a double to give 2.0
(char) 122      // converts 122 to a char whose code is 122
(unsigned short) 3.14   // gives 3 as an unsigned short
```

As shown by these examples, the built-in type identifiers can be used as **type operators**. Type operators are unary (i.e., take one operand) and appear inside brackets to the left of their operand. This is called **explicit type conversion**. When the type name is just one word, an alternate notation may be used in which the brackets appear around the operand:

```
int(3.14)       // same as: (int) 3.14
```

In some cases, C++ also performs **implicit type conversion**. This happens when values of different types are mixed in an expression. For example:

```
double  d = 1;          // d receives 1.0
int     i = 10.5;       // i receives 10
i = i + d;              // means: i = int(double(i) + d)
```

In the last example, `i + d` involves mismatching types, so `i` is first converted to double (*promoted*) and then added to `d`. The result is a `double` which does not match the type of `i` on the left side of the assignment, so it is converted to `int` (*demoted*) before being assigned to `i`.

The above rules represent some simple but common cases for type conversion. More complex cases will be examined later in the book after we have discussed other data types and classes.

□

Exercises

2.1 Write expressions for the following:

- To test if a number *n* is even.
- To test if a character *c* is a digit.
- To test if a character *c* is a letter.
- To do the test: *n* is odd and positive or *n* is even and negative.
- To set the *n*-th bit of a long integer *f* to 1.
- To reset the *n*-th bit of a long integer *f* to 0.
- To give the absolute value of a number *n*.
- To give the number of characters in a null-terminated string literal *s*.

2.2 Add extra brackets to the following expressions to explicitly show the order in which the operators are evaluated:

```
(n <= p + q && n >= p - q || n == 0)
(++n * q-- / ++p - q)
(n | p & q ^ p << 2 + q)
(p < q ? n < p ? q * n - 2 : q / n + 1 : q - n)
```

2.3 What will be the value of each of the following variables after its initialization:

```
double  d = 2 * int(3.14);
long    k = 3.14 - 3;
char    c = 'a' + 2;
char    c = 'p' + 'A' - 'a';
```

2.4 Write a program which inputs a positive integer *n* and outputs 2 raised to the power of *n*.

2.5 Write a program which inputs three numbers and outputs the message `Sorted` if the numbers are in ascending order, and outputs `Not sorted` otherwise.

□

3. Statements

This chapter introduces the various forms of C++ statements for composing programs. Statements represent the lowest-level building blocks of a program. Roughly speaking, each statement represents a computational step which has a certain **side-effect**. (A side-effect can be thought of as a change in the program state, such as the value of a variable changing because of an assignment.) Statements are useful because of the side-effects they cause, the combination of which enables the program to serve a specific purpose (e.g., sort a list of names).

A running program spends all of its time executing statements. The order in which statements are executed is called **flow control** (or control flow). This term reflect the fact that the currently executing statement has the *control* of the CPU, which when completed will be handed over (*flow*) to another statement. Flow control in a program is typically sequential, from one statement to the next, but may be diverted to other paths by branch statements. Flow control is an important consideration because it determines what is executed during a run and what is not, therefore affecting the overall outcome of the program.

Like many other procedural languages, C++ provides different forms of statements for different purposes. Declaration statements are used for defining variables. Assignment-like statements are used for simple, algebraic computations. Branching statements are used for specifying alternate paths of execution, depending on the outcome of a logical condition. Loop statements are used for specifying computations which need to be repeated until a certain logical condition is satisfied. Flow control statements are used to divert the execution path to another part of the program. We will discuss these in turn.

Simple and Compound Statements

A **simple** statement is a computation terminated by a semicolon. Variable definitions and semicolon-terminated expressions are examples:

```
int i;              // declaration statement
++i;                // this has a side-effect
double d = 10.5;    // declaration statement
d + 5;              // useless statement!
```

The last example represents a useless statement, because it has no side-effect (`d` is added to 5 and the result is just discarded).

The simplest statement is the null statement which consists of just a semicolon:

```
;       // null statement
```

Although the null statement has no side-effect, as we will see later in the chapter, it has some genuine uses.

Multiple statements can be combined into a **compound** statement by enclosing them within braces. For example:

```
{ int min, i = 10, j = 20;
  min = (i < j ? i : j);
  cout << min << '\n';
}
```

Compound statements are useful in two ways: (i) they allow us to put multiple statements in places where otherwise only single statements are allowed, and (ii) they allow us to introduce a new **scope** in the program. A scope is a part of the program text within which a variable remains defined. For example, the scope of `min`, `i`, and `j` in the above example is from where they are defined till the closing brace of the compound statement. Outside the compound statement, these variables are not defined.

Because a compound statement may contain variable definitions and defines a scope for them, it is also called a **block**. The scope of a C++ variable is limited to the block immediately enclosing it. Blocks and scope rules will be described in more detail when we discuss functions in the next chapter.

The if Statement

It is sometimes desirable to make the execution of a statement dependent upon a condition being satisfied. The `if` statement provides a way of expressing this, the general form of which is:

```
if (expression)
    statement;
```

First *expression* is evaluated. If the outcome is nonzero then *statement* is executed. Otherwise, nothing happens.

For example, when dividing two values, we may want to check that the denominator is nonzero:

```
if (count != 0)
    average = sum / count;
```

To make multiple statements dependent on the same condition, we can use a compound statement:

```
if (balance > 0) {
    interest = balance * creditRate;
    balance += interest;
}
```

A variant form of the if statement allows us to specify two alternative statements: one which is executed if a condition is satisfied and one which is executed if the condition is *not* satisfied. This is called the if-else statement and has the general form:

```
if (expression)
    statement1;
else
    statement2;
```

First *expression* is evaluated. If the outcome is nonzero then $statement_1$ is executed. Otherwise, $statement_2$ is executed.

For example:

```
if (balance > 0) {
    interest = balance * creditRate;
    balance += interest;
} else {
    interest = balance * debitRate;
    balance += interest;
}
```

Given the similarity between the two alternative parts, the whole statement can be simplified to:

```
if (balance > 0)
    interest = balance * creditRate;
else
    interest = balance * debitRate;
balance += interest;
```

Or simplified even further using a conditional expression:

```
interest = balance * (balance > 0 ? creditRate : debitRate);
balance += interest;
```

Or just:

```
balance += balance * (balance > 0 ? creditRate : debitRate);
```

If statements may be nested by having an if statement appear inside another if statement. For example:

```
if (callHour > 6) {
    if (callDuration <= 5)
        charge = callDuration * tarrif1;
    else
        charge = 5 * tarrif1 + (callDuration - 5) * tarrif2;
} else
    charge = flatFee;
```

A frequently-used form of nested if statements involves the else part consisting of another if-else statement. For example:

```
if (ch >= '0' && ch <= '9')
    kind = digit;
else {
    if (ch >= 'A' && ch <= 'Z')
        kind = upperLetter;
    else {
        if (ch >= 'a' && ch <= 'z')
            kind = lowerLetter;
        else
            kind = special;
    }
}
```

For improved readability, it is conventional to format such cases as follows:

```
if (ch >= '0' && ch <= '9')
    kind = digit;
else if (cha >= 'A' && ch <= 'Z')
    kind = capitalLetter;
else if (ch >= 'a' && ch <= 'z')
    kind = smallLetter;
else
    kind = special;
```

The switch Statement

The `switch` statement provides a way of choosing between a set of alternatives, based on the value of an expression. The general form of the switch statement is:

```
switch (expression) {
    case constant1:
        statements;
    ...
    case constantn:
        statements;
    default:
        statements;
}
```

First *expression* (called the switch **tag**) is evaluated, and the outcome is compared to each of the numeric *constant*s (called case **labels**), in the order they appear, until a match is found. The *statements* following the matching case are then executed. Note the plural: each case may be followed by zero or more statements (not just one statement). Execution continues until either a `break` statement is encountered or all intervening statements until the end of the switch statement are executed. The final `default` case is optional and is exercised if none of the earlier cases provide a match.

For example, suppose we have parsed a binary arithmetic operation into its three components and stored these in variables `operator`, `operand1`, and `operand2`. The following switch statement performs the operation and stored the result in `result`.

```
switch (operator) {
    case '+':   result = operand1 + operand2;
                break;
    case '-':   result = operand1 - operand2;
                break;
    case '*':   result = operand1 * operand2;
                break;
    case '/':   result = operand1 / operand2;
                break;
    default:    cout << "unknown operator: " << ch << '\n';
                break;
}
```

As illustrated by this example, it is usually necessary to include a break statement at the end of each case. The break terminates the switch statement by jumping to the very end of it. There are, however, situations in which it makes sense to have a case without a break. For example, if we extend the above statement to also allow `x` to be used as a multiplication operator, we will have:

```
switch (operator) {
```



```
        case '+':   result = operand1 + operand2;
                    break;
        case '-':   result = operand1 - operand2;
                    break;
        case 'x':
        case '*':   result = operand1 * operand2;
                    break;
        case '/':   result = operand1 / operand2;
                    break;
        default:    cout << "unknown operator: " << ch << '\n';
                    break;
    }
```

Because `case 'x'` has no break statement (in fact no statement at all!), when this case is satisfied, execution proceeds to the statements of the next case and the multiplication is performed.

It should be obvious that any switch statement can also be written as multiple if-else statements. The above statement, for example, may be written as:

```
if (operator == '+')
    result = operand1 + operand2;
else if (operator == '-')
    result = operand1 - operand2;
else if (operator == 'x' || operator == '*')
    result = operand1 * operand2;
else if (operator == '/')
    result = operand1 / operand2;
else
    cout << "unknown operator: " << ch << '\n';
```

However, the switch version is arguably neater in this case. In general, preference should be given to the switch version when possible. The if-else approach should be reserved for situation where a switch cannot do the job (e.g., when the conditions involved are not simple equality expressions, or when the case labels are not numeric constants).

The while Statement

The `while` statement (also called **while loop**) provides a way of repeating an statement while a condition holds. It is one of the three flavors of **iteration** in C++. The general form of the while statement is:

```
while (expression)
    statement;
```

First *expression* (called the **loop condition**) is evaluated. If the outcome is nonzero then *statement* (called the **loop body**) is executed and the whole process is repeated. Otherwise, the loop is terminated.

For example, suppose we wish to calculate the sum of all numbers from 1 to some integer denoted by *n*. This can be expressed as:

```
i = 1;
sum = 0;
while (i <= n)
    sum += i++;
```

For `n` set to 5, Table 3.1 provides a trace of the loop by listing the values of the variables involved and the loop condition.

Table 3.1 While loop trace.

Iteration	`i`	`n`	`i <= n`	`sum += i++`
First	1	5	1	1
Second	2	5	1	3
Third	3	5	1	6
Fourth	4	5	1	10
Fifth	5	5	1	15
Sixth	6	5	0	

It is not unusual for a while loop to have an empty body (i.e., a null statement). The following loop, for example, sets `n` to its greatest odd factor.

```
while (n % 2 == 0 && n /= 2)
    ;
```

Here the loop condition provides all the necessary computation, so there is no real need for a body. The loop condition not only tests that `n` is even, it also divides `n` by two and ensures that the loop will terminate should `n` be zero.

The do Statement

The `do` statement (also called **do loop**) is similar to the `while` statement, except that its body is executed first and then the loop condition is examined. The general form of the do statement is:

```
do
    statement;
while (expression);
```

First *statement* is executed and then *expression* is evaluated. If the outcome of the latter is nonzero then the whole process is repeated. Otherwise, the loop is terminated.

The do loop is less frequently used than the while loop. It is useful for situations where we need the loop body to be executed at least once, regardless of the loop condition. For example, suppose we wish to repeatedly read a value and print its square, and stop when the value is zero. This can be expressed as the following loop:

```
do {
    cin >> n;
    cout << n * n << '\n';
} while (n != 0);
```

Unlike the while loop, the do loop is never used in situations where it would have a null body. Although a do loop with a null body would be equivalent to a similar while loop, the latter is always preferred for its superior readability.

The for Statement

The `for` statement (also called **for loop**) is similar to the `while` statement, but has two additional components: an expression which is evaluated only once before everything else, and an expression which is evaluated once at the end of each iteration. The general form of the for statement is:

`for (`*expression*$_1$; *expression*$_2$; *expression*$_3$`)`
    statement;

First *expression*$_1$ is evaluated. Each time round the loop, *expression*$_2$ is evaluated. If the outcome is nonzero then statement is executed and *expression*$_3$ is evaluated. Otherwise, the loop is terminated. The general for loop is equivalent to the following while loop:

expression$_1$;
`while (`*expression*$_2$`) {`
    statement;
    expression$_3$;
`}`

The most common use of for loops is for situations where a variable is incremented or decremented with every iteration of the loop. The following for loop, for example, calculates the sum of all integers from 1 to `n`.

```
sum = 0;
for (i = 1; i <= n; ++i)
    sum += i;
```

This is preferred to the while-loop version we saw earlier. In this example, `i` is usually called the **loop variable**.

C++ allows the first expression in a for loop to be a variable definition. In the above loop, for example, `i` can be defined inside the loop itself:

```
for (int i = 1; i <= n; ++i)
    sum += i;
```

Contrary to what may appear, the scope for `i` is *not* the body of the loop, but the loop itself. Scope-wise, the above is equivalent to:

```
int i;
for (i = 1; i <= n; ++i)
    sum += i;
```


Any of the three expressions in a for loop may be empty. For example, removing the first and the third expression gives us something identical to a while loop:

```
for (; i != 0;)     // is equivalent to:  while (i != 0)
    something;      //                        something;
```

Removing all the expressions gives us an infinite loop. This loop's condition is assumed to be always true:

```
for (;;)                // infinite loop
    something;
```

For loops with multiple loop variables are not unusual. In such cases, the comma operator is used to separate their expressions:

```
for (i = 0, j = 0; i + j < n; ++i, ++j)
    something;
```

Because loops are statements, they can appear inside other loops. In other words, loops can be nested. For example,

```
for (int i = 1; i <= 3; ++i)
    for (int j = 1; j <= 3; ++j)
        cout << '(' << i << ',' << j << ")\n";
```

produces the product of the set {1,2,3} with itself, giving the output:

```
(1,1)
(1,2)
(1,3)
(2,1)
(2,2)
(2,3)
(3,1)
(3,2)
(3,3)
```

The continue Statement

The `continue` statement terminates the current iteration of a loop and instead jumps to the next iteration. It applies to the loop immediately enclosing the continue statement. It is an error to use the continue statement outside a loop.

In while and do loops, the next iteration commences from the loop condition. In a for loop, the next iteration commences from the loop's third expression. For example, a loop which repeatedly reads in a number, processes it but ignores negative numbers, and terminates when the number is zero, may be expressed as:

```
do {
    cin >> num;
    if (num < 0) continue;
    // process num here...
} while (num != 0);
```

This is equivalent to:

```
do {
    cin >> num;
    if (num >= 0) {
        // process num here...
    }
} while (num != 0);
```

A variant of this loop which reads in a number exactly *n* times (rather than until the number is zero) may be expressed as:

```
for (i = 0; i < n; ++i) {
    cin >> num;
    if (num < 0) continue;          // causes a jump to: ++i
    // process num here...
}
```

When the continue statement appears inside nested loops, it applies to the loop immediately enclosing it, and not to the outer loops. For example, in the following set of nested loops, the continue applies to the for loop, and not the while loop:

```
while (more) {
    for (i = 0; i < n; ++i) {
        cin >> num;
        if (num < 0) continue;          // causes a jump to: ++i
        // process num here...
    }
    //etc...
}
```


The break Statement

A `break` statement may appear inside a loop (while, do, or for) or a switch statement. It causes a jump out of these constructs, and hence terminates them. Like the continue statement, a break statement only applies to the loop or switch immediately enclosing it. It is an error to use the break statement outside a loop or a switch.

For example, suppose we wish to read in a user password, but would like to allow the user a limited number of attempts:

```
for (i = 0; i < attempts; ++i) {
    cout << "Please enter your password: ";
    cin >> password;
    if (Verify(password))   // check password for correctness
        break;              // drop out of the loop
    cout << "Incorrect!\n";
}
```

Here we have assumed that there is a function called `Verify` which checks a password and returns true if it is correct, and false otherwise.

Rewriting the loop without a break statement is always possible by using an additional logical variable (`verified`) and adding it to the loop condition:

```
verified = 0;
for (i = 0; i < attempts && !verified; ++i) {
    cout << "Please enter your password: ";
    cin >> password;
    verified = Verify(password));
    if (!verified)
        cout << "Incorrect!\n";
}
```

The break version is arguably simpler and therefore preferred.

The goto Statement

The `goto` statement provides the lowest-level of jumping. It has the general form:

`goto` *label*`;`

where *label* is an identifier which marks the jump destination of goto. The label should be followed by a colon and appear before a statement within the same function as the goto statement itself.

For example, the role of the break statement in the for loop in the previous section can be emulated by a goto:

```
    for (i = 0; i < attempts; ++i) {
        cout << "Please enter your password: ";
        cin >> password;
        if (Verify(password))   // check password for correctness
            goto out;           // drop out of the loop
        cout << "Incorrect!\n";
    }
out:
    //etc...
```

Because goto provides a free and unstructured form of jumping (unlike break and continue), it can be easily misused. Most programmers these days avoid using it altogether in favor of clear programming. Nevertheless, goto does have some legitimate (though rare) uses. Because of the potential complexity of such cases, furnishing of examples is postponed to the later parts of the book.

The return Statement

The `return` statement enables a function to return a value to its caller. It has the general form:

`return` *expression*`;`

where *expression* denotes the value returned by the function. The type of this value should match the return type of the function. For a function whose return type is `void`, *expression* should be empty:

```
return;
```

The only function we have discussed so far is `main`, whose return type is always `int`. The return value of main is what the program returns to the operating system when it completes its execution. Under UNIX, for example, it its conventional to return 0 from `main` when the program executes without errors. Otherwise, a non-zero error code is returned. For example:

```
int main (void)
{
    cout << "Hello World\n";
    return 0;
}
```

When a function has a non-void return value (as in the above example), failing to return a value will result in a compiler warning. The actual return value will be undefined in this case (i.e., it will be whatever value which happens to be in its corresponding memory location at the time).

Exercises

3.1 Write a program which inputs a person's height (in centimeters) and weight (in kilograms) and outputs one of the messages: `underweight`, `normal`, or `overweight`, using the criteria:

Underweight:	weight < height/2.5
Normal:	height/2.5 <= weight <= height/2.3
Overweight:	height/2.3 < weight

3.2 Assuming that `n` is 20, what will the following code fragment output when executed?

```
if (n >= 0)
    if (n < 10)
        cout << "n is small\n";
else
    cout << "n is negative\n";
```

3.3 Write a program which inputs a date in the format `dd/mm/yy` and outputs it in the format `month dd, year`. For example, `25/12/61` becomes:

```
December 25, 1961
```

3.4 Write a program which inputs an integer value, checks that it is positive, and outputs its factorial, using the formulas:

$$factorial(0) = 1$$
$$factorial(n) = n \times factorial(n\text{-}1)$$

3.5 Write a program which inputs an octal number and outputs its decimal equivalent. The following example illustrates the expected behavior of the program:

```
Input an octal number: 214
Octal(214) = Decimal(532)
```

3.6 Write a program which produces a simple multiplication table of the following format for integers in the range 1 to 9:

```
1 x 1 = 1
1 x 2 = 2
...
9 x 9 = 81
```


4. Functions

This chapter describes user-defined functions as one of the main building blocks of C++ programs. The other main building block — user-defined classes — will be discussed in Chapter 6.

A function provides a convenient way of packaging a computational recipe, so that it can be used as often as required. A **function definition** consists of two parts: interface and body. The **interface** of a function (also called its **prototype**) specifies how it may be used. It consists of three entities:

- The function **name**. This is simply a unique identifier.
- The function **parameters** (also called its **signature**). This is a set of zero or more typed identifiers used for passing values to and from the function.
- The function **return type**. This specifies the type of value the function returns. A function which returns nothing should have the return type `void`.

The **body** of a function contains the computational steps (statements) that comprise the function.

Using a function involves 'calling' it. A **function call** consists of the function name followed by the call operator brackets '`()`', inside which zero or more comma-separated **arguments** appear. The number of arguments should match the number of function parameters. Each argument is an expression whose type should match the type of the corresponding parameter in the function interface.

When a function call is executed, the arguments are first evaluated and their resulting values are assigned to the corresponding parameters. The function body is then executed. Finally, the function return value (if any) is passed to the caller.

Since a call to a function whose return type is non-`void` yields a return value, the call is an expression and may be used in other expressions. By contrast, a call to a function whose return type is `void` is a statement.

A Simple Function

Listing 4.1 shows the definition of a simple function which raises an integer to the power of another, positive integer.

Listing 4.1

```
1  int Power (int base, unsigned int exponent)
2  {
3      int result = 1;

4      for (int i = 0; i < exponent; ++i)
5          result *= base;
6      return result;
7  }
```

Annotation

1 This line defines the function interface. It starts with the return type of the function (`int` in this case). The function name appears next followed by its parameter list. `Power` has two parameters (`base` and `exponent`) which are of types `int` and `unsigned int`, respectively Note that the syntax for parameters is similar to the syntax for defining variables: type identifier followed by the parameter name. However, it is *not* possible to follow a type identifier with multiple comma-separated parameters:

```
int Power (int base, exponent)  // Wrong!
```

2 This brace marks the beginning of the function body.

3 This line is a **local** variable definition.

4-5 This for-loop raises `base` to the power of `exponent` and stores the outcome in `result`.

6 This line returns `result` as the return value of the function.

7 This brace marks the end of the function body.

Listing 4.2 illustrates how this function is called. The effect of this call is that first the argument values 2 and 8 are, respectively, assigned to the parameters `base` and `exponent`, and then the function body is evaluated.

Listing 4.2

```
1  #include <iostream.h>

2  main (void)
3  {
4      cout << "2 ^ 8 = " << Power(2,8) << '\n';
5  }
```

When run, this program will produce the following output:


```
2 ^ 8 = 256
```

In general, a function should be declared before its is used. A **function declaration** simply consists of the function prototype, which specifies the function name, parameter types, and return type. Line 2 in Listing 4.3 shows how `Power` may be declared for the above program. Although a function may be declared without its parameter names,

```
int Power (int, unsigned int);
```

this is not recommended unless the role of the parameters is obvious..

Listing 4.3

```
1  #include <iostream.h>

2  int Power (int base, unsigned int exponent); // function declaration

3  main (void)
4  {
5      cout << "2 ^ 8 = " << Power(2,8) << '\n';
6  }

7  int Power (int base, unsigned int exponent)
8  {
9      int result = 1;

10     for (int i = 0; i < exponent; ++i)
11         result *= base;
12     return result;
13 }
```

Because a function definition contains a prototype, it also serves as a declaration. Therefore if the definition of a function appears before its use, no additional declaration is needed. Use of function prototypes is nevertheless encouraged for all circumstances. Collecting these in a separate header file enables other programmers to quickly access the functions without having to read their entire definitions.

□

C++ supports two styles of parameters: value and reference. A **value parameter** receives a *copy* of the value of the argument passed to it. As a result, if the function makes any changes to the parameter, this will not affect the argument. For example, in

```
#include <iostream.h>

void Foo (int num)
{
    num = 0;
    cout << "num = " << num << '\n';
}

int main (void)
{
    int x = 10;

    Foo(x);
    cout << "x = " << x << '\n';
    return 0;
}
```

the single parameter of `Foo` is a value parameter. As far as this function is concerned, `num` behaves just like a local variable inside the function. When the function is called and `x` passed to it, `num` receives a copy of the value of `x`. As a result, although `num` is set to 0 by the function, this does not affect `x`. The program produces the following output:

```
num = 0;
x = 10;
```

A **reference parameter**, on the other hand, receives the argument passed to it and works on it directly. Any changes made by the function to a reference parameter is in effect directly applied to the argument. Reference parameters will be further discussed in Chapter 5.

Within the context of function calls, the two styles of passing arguments are, respectively, called **pass-by-value** and **pass-by-reference**. It is perfectly valid for a function to use pass-by-value for some of its parameters and pass-by-reference for others. The former is used much more often in practice.

□

Global and Local Scope

Everything defined at the program scope level (i.e., outside functions and classes) is said to have a **global scope**. Thus the sample functions we have seen so far all have a global scope. Variables may also be defined at the global scope:

```
int year = 1994;        // global variable
int Max (int, int);     // global function
int main (void)         // global function
{
    //...
}
```

Uninitialized global variables are automatically initialized to zero.

Since global entities are visible at the program level, they must also be unique at the program level. This means that the same global variable or function may not be *defined* more than once at the global level. (However, as we will see later, a function name may be reused so long as its signature remains unique.) Global entities are generally accessible everywhere in the program.

Each block in a program defines a **local scope**. Thus the body of a function represents a local scope. The parameters of a function have the same scope as the function body. Variables defined within a local scope are visible to that scope only. Hence, a variable need only be unique within its own scope. Local scopes may be nested, in which case the inner scopes override the outer scopes. For example, in

```
int xyz;                    // xyz is global
void Foo (int xyz)      // xyz is local to the body of Foo
{
    if (xyz > 0) {
        double xyz;     // xyz is local to this block
        //...
    }
}
```

there are three distinct scopes, each containing a distinct `xyz`.

Generally, the lifetime of a variable is limited to its scope. So, for example, global variables last for the duration of program execution, while local variables are created when their scope is entered and destroyed when their scope is exited. The memory space for global variables is reserved *prior* to program execution commencing, whereas the memory space for local variables is allocated on the fly *during* program execution.

□

Because a local scope overrides the global scope, having a local variable with the same name as a global variable makes the latter inaccessible to the local scope. For example, in

```
int error;

void Error (int error)
{
    //...
}
```

the global `error` is inaccessible inside `Error`, because it is overridden by the local `error` parameter.

This problem is overcome using the unary scope operator `::` which takes a global entity as argument:

```
int error;

void Error (int error)
{
    //...
    if (::error != 0)           // refers to global error
        //...
}
```

□

Auto Variables

Because the lifetime of a local variable is limited and is determined automatically, these variables are also called **automatic**. The storage class specifier `auto` may be used to explicitly specify a local variable to be automatic. For example:

```
void Foo (void)
{
    auto int xyz;       // same as: int xyz;
    //...
}
```

This is rarely used because all local variables are by default automatic.

□

Register Variables

As mentioned earlier, variables generally denote memory locations where variable values are stored. When the program code refers to a variable (e.g., in an expression), the compiler generates machine code which accesses the memory location denoted by the variable. For frequently-used variables (e.g., loop variables), efficiency gains can be obtained by keeping the variable in a register instead thereby avoiding memory access for that variable.

The storage class specifier `register` may be used to indicate to the compiler that the variable should be stored in a register if possible. For example:

```
for (register int i = 0; i < n; ++i)
    sum += i;
```

Here, each time round the loop, `i` is used three times: once when it is compared to `n`, once when it is added to `sum`, and once when it is incremented. Therefore it makes sense to keep `i` in a register for the duration of the loop.

Note that `register` is only a *hint* to the compiler, and in some cases the compiler may choose not to use a register when it is asked to do so. One reason for this is that any machine has a limited number of registers and it may be the case that they are all in use.

Even when the programmer does not use `register` declarations, many optimizing compilers try to make an intelligent guess and use registers where they are likely to improve the performance of the program.

Use of register declarations can be left as an after thought; they can always be added later by reviewing the code and inserting it in appropriate places.

□

Static Variables and Functions

It is often useful to confine the accessibility of a global variable or function to a single file. This is facilitated by the storage class specifier `static`. For example, consider a puzzle game program which consists of three files for game generation, game solution, and user interface. The game solution file would contain a `Solve` function and a number of other functions ancillary to `Solve`. Because the latter are only for the private use of `Solve`, it is best not to make them accessible outside the file:

```
static int FindNextRoute (void) // only accessible in this file
{
    //...
}
//...

int Solve (void)                // accessible outside this file
{
    //...
}
```

The same argument may be applied to the global variables in this file that are for the private use of the functions in the file. For example, a global variable which records the length of the shortest route so far is best defined as static:

```
static int shortestRoute;       // static global variable
```

A local variable in a function may also be defined as static. The variable will remain only accessible within its local scope; however, its lifetime will no longer be confined to this scope, but will instead be global. In other words, a static local variable is a global variable which is only accessible within its local scope.

Static local variables are useful when we want the value of a local variable to persist across the calls to the function in which it appears. For example, consider an `Error` function which keeps a count of the errors and aborts the program when the count exceeds a preset limit:

```
void Error (char *message)
{
    static int count = 0;       // static local variable

    if (++count > limit)
        Abort();
    //...
}
```

Like global variables, static local variables are automatically initialized to 0.

□

Extern Variables and Functions

Because a global variable may be defined in one file and referred to in other files, some means of telling the compiler that the variable is defined elsewhere may be needed. Otherwise, the compiler may object to the variable as undefined. This is facilitated by an `extern` declaration. For example, the declaration

```
extern int size;                    // variable declaration
```

informs the compiler that `size` is actually defined somewhere (may be later in this file or in another file). This is called a variable **declaration** (not definition) because it does not lead to any storage being allocated for `size`.

It is a poor programming practice to include an initializer for an `extern` variable, since this causes it to become a variable definition and have storage allocated for it:

```
extern int size = 10;               // no longer a declaration!
```

If there is another definition for `size` elsewhere in the program, it will eventually clash with this one.

Function prototypes may also be declared as extern, but this has no effect when a prototype appears at the global scope. It is more useful for declaring function prototypes inside a function. For example:

```
double Tangent (double angle)
{
    extern  double sin(double);     // defined elsewhere
    extern  double cos(double);     // defined elsewhere

    return sin(angle) / cos(angle);
}
```

The best place for `extern` declarations is usually in header files so that they can be easily included and shared by source files.

□

Symbolic Constants

Preceding a variable definition by the keyword `const` makes that variable read-only (i.e., a symbolic constant). A constant must be initialized to some value when it is defined. For example:

```
const int    maxSize = 128;
const double pi = 3.141592654;
```

Once defined, the value of a constant cannot be changed:

```
maxSize = 256;                  // illegal!
```

A constant with no type specifier is assumed to be of type `int`:

```
const maxSize = 128;            // maxSize is of type int
```

With pointers, two aspects need to be considered: the pointer itself, and the object pointed to, either of which or both can be constant:

```
const char *str1 = "pointer to constant";
char *const str2 = "constant pointer";
const char *const str3 = "constant pointer to constant";
str1[0] = 'P';                  // illegal!
str1 = "ptr to const";          // ok
str2 = "const ptr";             // illegal!
str2[0] = 'P';                  // ok
str3 = "const to const ptr";    // illegal!
str3[0] = 'C';                  // illegal!
```

A function parameter may also be declared to be constant. This may be used to indicate that the function does not change the value of a parameter:

```
int Power (const int base, const unsigned int exponent)
{
    //...
}
```

A function may also return a constant result:

```
const char* SystemVersion (void)
{
    return "5.2.1";
}
```

The usual place for constant definition is within header files so that they can be shared by source files.

□

Enumerations

An enumeration of symbolic constants is introduced by an `enum` declaration. This is useful for declaring a set of closely-related constants. For example,

```
enum {north, south, east, west};
```

introduces four **enumerators** which have integral values starting from 0 (i.e., `north` is 0, `south` is 1, etc.) Unlike symbolic constants, however, which are read-only variables, enumerators have no allocated memory.

The default numbering of enumerators can be overruled by explicit initialization:

```
enum {north = 10, south, east = 0, west};
```

Here, `south` is 11 and `west` is 1.

An enumeration can also be named, where the name becomes a user-defined type. This is useful for defining variables which can only be assigned a limited set of values. For example, in

```
enum Direction {north, south, east, west};
Direction d;
```

`d` can only be assigned one of the enumerators for `Direction`.

Enumerations are particularly useful for naming the cases of a switch statement.

```
switch (d) {
    case north: //...
    case south: //...
    case east:  //...
    case west:  //...
}
```

We will extensively use the following enumeration for representing boolean values in the programs in this book:

```
enum Bool {false, true};
```

□

Runtime Stack

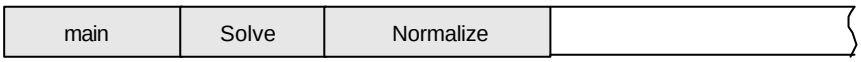
Like many other modern programming languages, C++ function call execution is based on a runtime stack. When a function is called, memory space is allocated on this stack for the function parameters, return value, and local variables, as well as a local stack area for expression evaluation. The allocated space is called a **stack frame**. When a function returns, the allocated stack frame is released so that it can be reused.

For example, consider a situation where `main` calls a function called `Solve` which in turn calls another function called `Normalize`:

```
int Normalize (void)
{
    //...
}

int Solve (void)
{
    //...
    Normalize();
    //...
}

int main (void)
{
    //...
    Solve();
    //...
}
```

Figure 4.1 illustrates the stack frame when `Normalize` is being executed.

Figure 4.1 Function call stack frames.



It is important to note that the calling of a function involves the overheads of creating a stack frame for it and removing the stack frame when it returns. For most functions, this overhead is negligible compared to the actual computation the function performs.

□

Inline Functions

Suppose that a program frequently requires to find the absolute value of an integer quantity. For a value denoted by `n`, this may be expressed as:

```
(n > 0 ? n : -n)
```

However, instead of replicating this expression in many places in the program, it is better to define it as a function:

```
int Abs (int n)
{
    return n > 0 ? n : -n;
}
```

The function version has a number of advantages. First, it leads to a more readable program. Second, it is reusable. And third, it avoid undesirable side-effects when the argument is itself an expression with side-effects.

The disadvantage of the function version, however, is that its frequent use can lead to a considerable performance penalty due to the overheads associated with calling a function. For example, if `Abs` is used within a loop which is iterated thousands of times, then it will have an impact on performance. The overhead can be avoided by defining `Abs` as an `inline` function:

```
inline int Abs (int n)
{
    return n > 0 ? n : -n;
}
```

The effect of this is that when `Abs` is called, the compiler, instead of generating code to call `Abs`, expands and substitutes the body of `Abs` in place of the call. While essentially the same computation is performed, no function call is involved and hence no stack frame is allocated.

Because calls to an inline function are expanded, no trace of the function itself will be left in the compiled code. Therefore, if a function is defined inline in one file, it may not be available to other files. Consequently, inline functions are commonly placed in header files so that they can be shared.

Like the `register` keyword, `inline` is a *hint* which the compiler is not obliged to observe. Generally, the use of inline should be restricted to simple, frequently used functions. A function which contains anything more than a couple of statements is unlikely to be a good candidate. Use of inline for excessively long and complex functions is almost certainly ignored by the compiler. □

Recursion

A function which calls itself is said to be **recursive**. Recursion is a general programming technique applicable to problems which can be defined in terms of themselves. Take the factorial problem, for instance, which is defined as:

- Factorial of 0 is 1.
- Factorial of a positive number *n* is *n* times the factorial of *n*-1.

The second line clearly indicates that factorial is defined in terms of itself and hence can be expressed as a recursive function:

```
int Factorial (unsigned int n)
{
    return n == 0 ? 1 : n * Factorial(n-1);
}
```

For `n` set to 3, Table 4.1 provides a trace of the calls to `Factorial`. The stack frames for these calls appear sequentially on the runtime stack, one after the other.

Table 4.1 Factorial(3) execution trace.

Call	`n`	`n == 0`	`n * Factorial(n-1)`	Returns
First	3	0	`3 * Factorial(2)`	6
Second	2	0	`2 * Factorial(1)`	2
Third	1	0	`1 * Factorial(0)`	1
Fourth	0	1		1

A recursive function must have at least one **termination condition** which can be satisfied. Otherwise, the function will call itself indefinitely until the runtime stack overflows. The Factorial function, for example, has the termination condition `n == 0` which, when satisfied, causes the recursive calls to fold back. (Note that for a negative `n` this condition will never be satisfied and `Factorial` will fail).

As a general rule, all recursive functions can be rewritten using iteration. In situations where the number of stack frames involved may be quite large, the iterative version is preferred. In other cases, the elegance and simplicity of the recursive version may give it the edge.

For factorial, for example, a very large argument will lead to as many stack frames. An iterative version is therefore preferred in this case:

```
int Factorial (unsigned int n)
{
    int result = 1;
    while (n > 0) result *= n--;
    return result;
}
```

□

Default Arguments

Default argument is a programming convenience which removes the burden of having to specify argument values for all of a function's parameters. For example, consider a function for reporting errors:

```
void Error (char *message, int severity = 0);
```

Here, `severity` has a default argument of 0; both the following calls are therefore valid:

```
Error("Division by zero", 3);    // severity set to 3
Error("Round off error");        // severity set to 0
```

As the first call illustrates, a default argument may be overridden by explicitly specifying an argument.

Default arguments are suitable for situations where certain (or all) function parameters frequently take the same values. In `Error`, for example, severity 0 errors are more common than others and therefore a good candidate for default argument. A less appropriate use of default arguments would be:

```
int Power (int base, unsigned int exponent = 1);
```

Because 1 (or any other value) is unlikely to be a frequently-used one in this situation.

To avoid ambiguity, all default arguments must be trailing arguments. The following declaration is therefore illegal:

```
void Error (char *message = "Bomb", int severity);   // Illegal!
```

A default argument need not necessarily be a constant. Arbitrary expressions can be used, so long as the variables used in the expression are available to the scope of the function definition (e.g., global variables).

The accepted convention for default arguments is to specify them in function declarations, not function definitions. Because function declarations appear in header files, this enables the user of a function to have control over the default arguments. Thus different default arguments can be specified for different situations. It is, however, illegal to specify two different default arguments for the same function in a file.

□

Variable Number of Arguments

It is sometimes desirable, if not necessary, to have functions which take a variable number of arguments. A simple example is a function which takes a set of menu options as arguments, displays the menu, and allows the user to choose one of the options. To be general, the function should be able to accept any number of options as arguments. This may be expressed as

```
int Menu (char *option1 ...);
```

which states that `Menu` should be given one argument or more.

`Menu` can access its arguments using a set of macro definitions in the header file `stdarg.h`, as illustrated by Listing 4.4. The relevant macros are highlighted in **bold**.

Listing 4.4

```
 1 #include <iostream.h>
 2 #include <stdarg.h>

 3 int Menu (char *option1 ...)
 4 {
 5     va_list args;             // argument list
 6     char*   option = option1;
 7     int     count = 0, choice = 0;

 8     va_start(args, option1);      // initialize args

 9     do {
10         cout << ++count << ". " << option << '\n';
11     } while ((option = va_arg(args, char*)) != 0);

12     va_end(args);             // clean up args
13     cout << "option? ";
14     cin >> choice;
15     return (choice > 0 && choice <= count) ? choice : 0;
16 }
```

Annotation

5 To access the arguments, `args` is declared to be of type `va_list`.

8 `Args` is initialized by calling `va_start`. The second argument to `va_start` must be the last function parameter explicitly declared in the function header (i.e., `option1` here).

11 Subsequent arguments are retrieved by calling `va_arg`. The second argument to `va_arg` must be the expected type of that argument (i.e., `char*` here). For this technique to work, the last argument must be a 0, marking the end of the argument list. `Va_arg` is called repeatedly until this 0 is reached.

12 Finally, va_end is called to restore the runtime stack (which may have been modified by the earlier calls).

The sample call

```
int n = Menu(
        "Open file",
        "Close file",
        "Revert to saved file",
        "Delete file",
        "Quit application",
        0);
```

will produce the following output:

```
1. Open file
2. Close file
3. Revert to saved file
4. Delete file
5. Quit application
option?
```

□

Command Line Arguments

When a program is executed under an operating system (such as DOS or UNIX), it can be passed zero or more arguments. These arguments appear after the program executable name and are separated by blanks. Because they appear on the same line as where operating system commands are issued, they are called **command line arguments**.

As an example, consider a program named `sum` which prints out the sum of a set of numbers provided to it as command line arguments. Dialog 4.1 illustrates how two numbers are passed as arguments to sum (`$` is the UNIX prompt).

Dialog 4.1

```
1  $ sum 10.4 12.5
2  22.9
3  $
```

Command line arguments are made available to a C++ program via the `main` function. There are two ways in which `main` can be defined:

```
int main (void);
int main (int argc, const char* argv[]);
```

The latter is used when the program is intended to accept command line arguments. The first parameter, `argc`, denotes the number of arguments passed to the program (including the name of the program itself). The second parameter, `argv`, is an array of the string constants which represent the arguments. For example, given the command line in Dialog 4.1, we have:

```
argc      is   3
argv[0]   is   "sum"
argv[1]   is   "10.4"
argv[2]   is   "12.5"
```

Listing 4.5 illustrates a simple implementation for `sum`. Strings are converted to real numbers using `atof`, which is defined in `stdlib.h`.

Listing 4.5

```
1  #include <iostream.h>
2  #include <stdlib.h>

3  int main (int argc, const char *argv[])
4  {
5      double sum = 0;
6      for (int i = 1; i < argc; ++i)
7          sum += atof(argv[i]);
8      cout << sum << '\n';
9      return 0;
10 }
```

□

Exercises

4.1 Write the programs in exercises 1.1 and 3.1 as functions.

4.2 Given the following definition of a `Swap` function

```
void Swap (int x, int y)
{
    int temp = x;
    x = y;
    y = temp;
}
```

what will be the value of `x` and `y` after the following call:

```
x = 10;
y = 20;
Swap(x, y);
```

4.3 What will the following program output when executed?

```
#include <iostream.h>
char *str = "global";

void Print (char *str)
{
    cout << str << '\n';
    {
        char *str = "local";
        cout << str << '\n';
        cout << ::str << '\n';
    }
    cout << str << '\n';
}

int main (void)
{
    Print("Parameter");
    return 0;
}
```

4.4 Write a function which outputs all the prime numbers between 2 and a given positive integer *n*:

```
void Primes (unsigned int n);
```

A number is prime if it is only divisible by itself and 1.

4.5 Define an enumeration called `Month` for the months of the year and use it to define a function which takes a month as argument and returns it as a constant string.

4.6 Define an inline function called `IsAlpha` which returns nonzero when its argument is a letter, and zero otherwise.

4.7 Define a recursive version of the `Power` function described in this chapter.

4.8 Write a function which returns the sum of a list of real values

```
double Sum (int n, double val ...);
```

where `n` denotes the number of values in the list.

□

5. Arrays, Pointers, and References

This chapter introduces the array, pointer, and reference data types and illustrates their use for defining variables.

An **array** consists of a set of objects (called its **elements**), all of which are of the same type and are arranged contiguously in memory. In general, only the array itself has a symbolic name, not its elements. Each element is identified by an **index** which denotes the position of the element in the array. The number of elements in an array is called its **dimension**. The dimension of an array is fixed and predetermined; it cannot be changed during program execution.

Arrays are suitable for representing composite data which consist of many similar, individual items. Examples include: a list of names, a table of world cities and their current temperatures, or the monthly transactions for a bank account.

A **pointer** is simply the address of an object in memory. Generally, objects can be accessed in two ways: directly by their symbolic name, or indirectly through a pointer. The act of getting to an object via a pointer to it, is called **dereferencing** the pointer. Pointer variables are defined to point to objects of a specific type so that when the pointer is dereferenced, a typed object is obtained.

Pointers are useful for creating **dynamic** objects during program execution. Unlike normal (global and local) objects which are allocated storage on the runtime stack, a dynamic object is allocated memory from a different storage area called the **heap**. Dynamic objects do not obey the normal scope rules. Their scope is explicitly controlled by the programmer.

A **reference** provides an alternative symbolic name (**alias**) for an object. Accessing an object through a reference is exactly the same as accessing it through its original name. References offer the power of pointers and the convenience of direct access to objects. They are used to support the call-by-reference style of function parameters, especially when large objects are being passed to functions.

Arrays

An array variable is defined by specifying its dimension and the type of its elements. For example, an array representing 10 height measurements (each being an integer quantity) may be defined as:

```
int heights[10];
```

The individual elements of the array are accessed by indexing the array. The first array element always has the index 0. Therefore, `heights[0]` and `heights[9]` denote, respectively, the first and last element of `heights`. Each of `heights` elements can be treated as an integer variable. So, for example, to set the third element to 177, we may write:

```
heights[2] = 177;
```

Attempting to access a nonexistent array element (e.g., `heights[-1]` or `heights[10]`) leads to a serious runtime error (called 'index out of bounds' error).

Processing of an array usually involves a loop which goes through the array element by element. Listing 5.1 illustrates this using a function which takes an array of integers and returns the average of its elements.

Listing 5.1

```
1  const int size = 3;

2  double Average (int nums[size])
3  {
4      double average = 0;

5      for (register i = 0; i < size; ++i)
6          average += nums[i];
7      return average/size;
8  }
```

Like other variables, an array may have an initializer. Braces are used to specify a list of comma-separated initial values for array elements. For example,

```
int nums[3] = {5, 10, 15};
```

initializes the three elements of `nums` to 5, 10, and 15, respectively. When the number of values in the initializer is less than the number of elements, the remaining elements are initialized to zero:

```
int nums[3] = {5, 10};      // nums[2] initializes to 0
```


When a complete initializer is used, the array dimension becomes redundant, because the number of elements is implicit in the initializer. The first definition of `nums` can therefore be equivalently written as:

```
int nums[] = {5, 10, 15};    // no dimension needed
```

Another situation in which the dimension can be omitted is for an array function parameter. For example, the `Average` function above can be improved by rewriting it so that the dimension of `nums` is not fixed to a constant, but specified by an additional parameter. Listing 5.2 illustrates this.

Listing 5.2

```
1  double Average (int nums[], int size)
2  {
3      double average = 0;

4      for (register i = 0; i < size; ++i)
5          average += nums[i];
6      return average/size;
7  }
```

A C++ string is simply an array of characters. For example,

```
char    str[] = "HELLO";
```

defines `str` to be an array of six characters: five letters and a null character. The terminating null character is inserted by the compiler. By contrast,

```
char    str[] = {'H', 'E', 'L', 'L', 'O'};
```

defines `str` to be an array of five characters.

It is easy to calculate the dimension of an array using the sizeof operator. For example, given an array `ar` whose element type is `Type`, the dimension of `ar` is:

```
sizeof(ar) / sizeof(Type)
```

□

Multidimensional Arrays

An array may have more than one dimension (i.e., two, three, or higher). The organization of the array in memory is still the same (a contiguous sequence of elements), but the programmer's perceived organization of the elements is different. For example, suppose we wish to represent the average seasonal temperature for three major Australian capital cities (see Table 5.1).

Table 5.1 Average seasonal temperature.

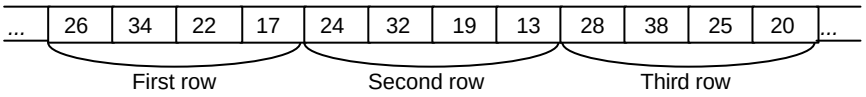
	Spring	Summer	Autumn	Winter
Sydney	26	34	22	17
Melbourne	24	32	19	13
Brisbane	28	38	25	20

This may be represented by a two-dimensional array of integers:

```
int seasonTemp[3][4];
```

The organization of this array in memory is as 12 consecutive integer elements. The programmer, however, can imagine it as three rows of four integer entries each (see Figure 5.1).

Figure 5.1 Organization of seasonTemp in memory.



As before, elements are accessed by indexing the array. A separate index is needed for each dimension. For example, Sydney's average summer temperature (first row, second column) is given by `seasonTemp[0][1]`.

The array may be initialized using a nested initializer:

```
int seasonTemp[3][4] = {
    {26, 34, 22, 17},
    {24, 32, 19, 13},
    {28, 38, 25, 20}
};
```

Because this is mapped to a one-dimensional array of 12 elements in memory, it is equivalent to:

```
int seasonTemp[3][4] = {
    26, 34, 22, 17, 24, 32, 19, 13, 28, 38, 25, 20
};
```

The nested initializer is preferred because as well as being more informative, it is more versatile. For example, it makes it possible to initialize only the first element of each row and have the rest default to zero:

```
int seasonTemp[3][4] = {{26}, {24}, {28}};
```

We can also omit the first dimension (but not subsequent dimensions) and let it be derived from the initializer:

```
int seasonTemp[][4] = {
    {26, 34, 22, 17},
    {24, 32, 19, 13},
    {28, 38, 25, 20}
};
```

Processing a multidimensional array is similar to a one-dimensional array, but uses nested loops instead of a single loop. Listing 5.3 illustrates this by showing a function for finding the highest temperature in `seasonTemp`.

Listing 5.3

```
1  const int rows      = 3;
2  const int columns   = 4;

3  int seasonTemp[rows][columns] = {
4      {26, 34, 22, 17},
5      {24, 32, 19, 13},
6      {28, 38, 25, 20}
7  };

8  int HighestTemp (int temp[rows][columns])
9  {
10     int highest = 0;

11     for (register i = 0; i < rows; ++i)
12         for (register j = 0; j < columns; ++j)
13             if (temp[i][j] > highest)
14                     highest = temp[i][j];
15     return highest;
16 }
```

□

Pointers

A pointer is simply the *address* of a memory location and provides an indirect way of accessing data in memory. A pointer variable is defined to 'point to' data of a specific type. For example:

```
int     *ptr1;      // pointer to an int
char    *ptr2;      // pointer to a char
```

The *value* of a pointer variable is the address to which it points. For example, given the definitions

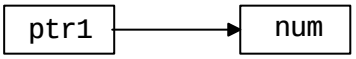
```
int     num;
```

we can write:

```
ptr1 = &num;
```

The symbol `&` is the **address** operator; it takes a variable as argument and returns the memory address of that variable. The effect of the above assignment is that the address of `num` is assigned to `ptr1`. Therefore, we say that `ptr1` points to `num`. Figure 5.2 illustrates this diagrammatically.

Figure 5.2 A simple integer pointer.



Given that `ptr1` points to `num`, the expression

```
*ptr1
```

dereferences `ptr1` to get to what it points to, and is therefore equivalent to `num`. The symbol `*` is the **dereference** operator; it takes a pointer as argument and returns the contents of the location to which it points.

In general, the type of a pointer must match the type of the data it is set to point to. A pointer of type `void*`, however, will match any type. This is useful for defining pointers which may point to data of different types, or whose type is originally unknown.

A pointer may be **cast** (type converted) to another type. For example,

```
ptr2 = (char*) ptr1;
```

converts `ptr1` to `char` pointer before assigning it to `ptr2`.

Regardless of its type, a pointer may be assigned the value 0 (called the **null** pointer). The null pointer is used for initializing pointers, and for marking the end of pointer-based data structures (e.g., linked lists).

□

Dynamic Memory

In addition to the program stack (which is used for storing global variables and stack frames for function calls), another memory area, called the **heap**, is provided. The heap is used for dynamically allocating memory blocks during program execution. As a result, it is also called **dynamic memory**. Similarly, the program stack is also called **static memory**.

Two operators are used for allocating and deallocating memory blocks on the heap. The `new` operator takes a type as argument and allocated a memory block for an object of that type. It returns a pointer to the allocated block. For example,

```
int  *ptr = new int;
char *str = new char[10];
```

allocate, respectively, a block for storing a single integer and a block large enough for storing an array of 10 characters.

Memory allocated from the heap does not obey the same scope rules as normal variables. For example, in

```
void Foo (void)
{
    char *str = new char[10];
    //...
}
```

when `Foo` returns, the local variable `str` is destroyed, but the memory block pointed to by `str` is *not*. The latter remains allocated until explicitly released by the programmer.

The `delete` operator is used for releasing memory blocks allocated by `new`. It takes a pointer as argument and releases the memory block to which it points. For example:

```
delete ptr;         // delete an object
delete [] str;      // delete an array of objects
```

Note that when the block to be deleted is an array, an additional `[]` should be included to indicate this. The significance of this will be explained later when we discuss classes.

Should `delete` be applied to a pointer which points to anything but a dynamically-allocated object (e.g., a variable on the stack), a serious runtime error may occur. It is harmless to apply `delete` to the 0 pointer.

Dynamic objects are useful for creating data which last beyond the function call which creates them. Listing 5.4 illustrates this using a function which takes a string parameter and returns a *copy* of the string.

Listing 5.4

```
1  #include <string.h>

2  char* CopyOf (const char *str)
3  {
4      char *copy = new char[strlen(str) + 1];

5      strcpy(copy, str);
6      return copy;
7  }
```

Annotation

1 This is the standard string header file which declares a variety of functions for manipulating strings.

4 The `strlen` function (declared in `string.h`) counts the characters in its string argument up to (but excluding) the final null character. Because the null character is not included in the count, we add 1 to the total and allocate an array of characters of that size.

5 The `strcpy` function (declared in `string.h`) copies its second argument to its first, character by character, including the final null character.

Because of the limited memory resources, there is always the possibility that dynamic memory may be exhausted during program execution, especially when many large blocks are allocated and none released. Should `new` be unable to allocate a block of the requested size, it will return 0 instead. It is the responsibility of the programmer to deal with such possibilities. The exception handling mechanism of C++ (explained in Chapter 10) provides a practical method of dealing with such problems.

□

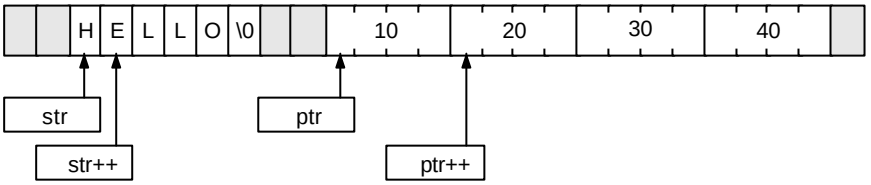
Pointer Arithmetic

In C++ one can add an integer quantity to or subtract an integer quantity from a pointer. This is frequently used by programmers and is called pointer arithmetic. Pointer arithmetic is *not* the same as integer arithmetic, because the outcome depends on the size of the object pointed to. For example, suppose that an `int` is represented by 4 bytes. Now, given

```
char *str = "HELLO";
int  nums[] = {10, 20, 30, 40};
int  *ptr = &nums[0];           // pointer to first element
```

`str++` advances `str` by one `char` (i.e., one byte) so that it points to the second character of `"HELLO"`, whereas `ptr++` advances `ptr` by one `int` (i.e., four bytes) so that it points to the second element of `nums`. Figure 5.3 illustrates this diagrammatically.

Figure 5.3 Pointer arithmetic.



It follows, therefore, that the elements of `"HELLO"` can be referred to as `*str`, `*(str + 1)`, `*(str + 2)`, etc. Similarly, the elements of `nums` can be referred to as `*ptr`, `*(ptr + 1)`, `*(ptr + 2)`, and `*(ptr + 3)`.

Another form of pointer arithmetic allowed in C++ involves subtracting two pointers of the same type. For example:

```
int *ptr1 = &nums[1];
int *ptr2 = &nums[3];
int n = ptr2 - ptr1;     // n becomes 2
```

Pointer arithmetic is very handy when processing the elements of an array. Listing 5.5 shows as an example a string copying function similar to `strcpy`.

Listing 5.5

```
1  void CopyString (char *dest, char *src)
2  {
3      while (*dest++ = *src++)
4          ;
5  }
```

Annotation

3 The condition of this loop assigns the contents of `src` to the contents of `dest` and then increments both pointers. This condition becomes 0 when the final null character of `src` is copied to `dest`.

In turns out that an array variable (such as `nums`) is itself the address of the first element of the array it represents. Hence the elements of `nums` can also be referred to using pointer arithmetic on `nums`, that is, `nums[i]` is equivalent to `*(nums + i)`. The difference between `nums` and `ptr` is that `nums` is a constant, so it cannot be made to point to anything else, whereas `ptr` is a variable and can be made to point to any other integer.

Listing 5.6 shows how the `HighestTemp` function (shown earlier in Listing 5.3) can be improved using pointer arithmetic.

Listing 5.6

```
1  int HighestTemp (const int *temp, const int rows, const int columns)
2  {
3      int highest = 0;

4      for (register i = 0; i < rows; ++i)
5          for (register j = 0; j < columns; ++j)
6              if (*(temp + i * columns + j) > highest)
7                  highest = *(temp + i * columns + j);
8      return highest;
9  }
```

Annotation

1 Instead of passing an array to the function, we pass an `int` pointer and two additional parameters which specify the dimensions of the array. In this way, the function is not restricted to a specific array size.

6 The expression `*(temp + i * columns + j)` is equivalent to `temp[i][j]` in the previous version of this function.

`HighestTemp` can be simplified even further by treating `temp` as a one-dimensional array of `row * column` integers. This is shown in Listing 5.7.

Listing 5.7

```
1  int HighestTemp (const int *temp, const int rows, const int columns)
2  {
3      int highest = 0;

4      for (register i = 0; i < rows * columns; ++i)
5          if (*(temp + i) > highest)
6              highest = *(temp + i);
7      return highest;
8  }
```

□

Function Pointers

It is possible to take the address of a function and store it in a function pointer. The pointer can then be used to indirectly call the function. For example,

```
int (*Compare)(const char*, const char*);
```

defines a function pointer named `Compare` which can hold the address of any function that takes two constant character pointers as arguments and returns an integer. The string comparison library function `strcmp`, for example, is such. Therefore:

```
Compare = &strcmp;          // Compare points to strcmp function
```

The & operator is not necessary and can be omitted:

```
Compare = strcmp;           // Compare points to strcmp function
```

Alternatively, the pointer can be defined and initialized at once:

```
int (*Compare)(const char*, const char*) = strcmp;
```

When a function address is assigned to a function pointer, the two types must match. The above definition is valid because `strcmp` has a matching function prototype:

```
int strcmp(const char*, const char*);
```

Given the above definition of `Compare`, `strcmp` can be either called directly, or indirectly via `Compare`. The following three calls are equivalent:

```
strcmp("Tom", "Tim");           // direct call
(*Compare)("Tom", "Tim");       // indirect call
Compare("Tom", "Tim");          // indirect call (abbreviated)
```

A common use of a function pointer is to pass it as an argument to another function; typically because the latter requires different versions of the former in different circumstances. A good example is a binary search function for searching through a sorted array of strings. This function may use a comparison function (such as `strcmp`) for comparing the search string against the array strings. This might not be appropriate for all cases. For example, `strcmp` is case-sensitive. If we wanted to do the search in a non-case-sensitive manner then a different comparison function would be needed.

As shown in Listing 5.8, by making the comparison function a parameter of the search function, we can make the latter independent of the former.

Listing 5.8

```
 1  int BinSearch (char *item, char *table[], int n,
 2                 int (*Compare)(const char*, const char*))
 3  {
 4      int bot = 0;
 5      int top = n - 1;
 6      int mid, cmp;

 7      while (bot <= top) {
 8          mid = (bot + top) / 2;
 9          if ((cmp = Compare(item,table[mid])) == 0)
10              return mid;             // return item index
11          else if (cmp < 0)
12              top = mid - 1;          // restrict search to lower half
13          else
14              bot = mid + 1;          // restrict search to upper half
15      }
16      return -1;                      // not found
17  }
```

Annotation

1 Binary search is a well-known algorithm for searching through a *sorted* list of items. The search list is denoted by `table` which is an array of strings of dimension `n`. The search item is denoted by `item`.

2 `Compare` is the function pointer to be used for comparing `item` against the array elements.

7 Each time round this loop, the search span is reduced by half. This is repeated until the two ends of the search span (denoted by `bot` and `top`) collide, or until a match is found.

9 The item is compared against the middle item of the array.

10 If `item` matches the middle item, the latter's index is returned.

11 If `item` is less than the middle item, then the search is restricted to the lower half of the array.

14 If `item` is greater than the middle item, then the search is restricted to the upper half of the array.

16 Returns -1 to indicate that there was no matching item.

The following example shows how `BinSearch` may be called with `strcmp` passed as the comparison function:

```
char *cities[] = {"Boston", "London", "Sydney", "Tokyo"};
cout << BinSearch("Sydney", cities, 4, strcmp) << '\n';
```

This will output 2 as expected. □

References

A reference introduces an **alias** for an object. The notation for defining references is similar to that of pointers, except that `&` is used instead of `*`. For example,

```
double num1 = 3.14;
double &num2 = num1;     // num is a reference to num1
```

defines `num2` as a reference to `num1`. After this definition `num1` and `num2` both refer to the same object, as if they were the same variable. It should be emphasized that a reference does not create a copy of an object, but merely a symbolic alias for it. Hence, after

```
num1 = 0.16;
```

both `num1` and `num2` will denote the value 0.16.

A reference must always be initialized when it is defined: it should be an alias for something. It would be illegal to define a reference and initialize it later.

```
double &num3;        // illegal: reference without an initializer
num3 = num1;
```

You can also initialize a reference to a constant. In this case a *copy* of the constant is made (after any necessary type conversion) and the reference is set to refer to the copy.

```
int &n = 1;          // n refers to a copy of 1
```

The reason that `n` becomes a reference to a copy of 1 rather than 1 itself is safety. Consider what could happen if this were not the case.

```
int &x = 1;
++x;
int y = x + 1;
```

The 1 in the first and the 1 in the third line are likely to be the same object (most compilers do constant optimization and allocate both 1's in the same memory location). So although we expect `y` to be 3, it could turn out to be 4. However, by forcing `x` to be a copy of 1, the compiler guarantees that the object denoted by `x` will be different from both 1's.

The most common use of references is for function parameters. Reference parameters facilitates the **pass-by-reference** style of arguments, as opposed to the **pass-by-value** style which we have used so far. To observe the differences, consider the three swap functions in Listing 5.9.

Listing 5.9

```
 1  void Swap1 (int x, int y)       // pass-by-value (objects)
 2  {
 3      int temp = x;
 4      x = y;
 5      y = temp;
 6  }

 7  void Swap2 (int *x, int *y)     // pass-by-value (pointers)
 8  {
 9      int temp = *x;
10      *x = *y;
11      *y = temp;
12  }

13  void Swap3 (int &x, int &y)     // pass-by-reference
14  {
15      int temp = x;
16      x = y;
17      y = temp;
18  }
```

Annotation

1 Although `Swap1` swaps `x` and `y`, this has no effect on the arguments passed to the function, because `Swap1` receives a *copy* of the arguments. What happens to the copy does not affect the original.

7 `Swap2` overcomes the problem of `Swap1` by using pointer parameters instead. By dereferencing the pointers, `Swap2` gets to the original values and swaps them.

13 `Swap3` overcomes the problem of `Swap1` by using reference parameters instead. The parameters become aliases for the arguments passed to the function and therefore swap them as intended.

`Swap3` has the added advantage that its call syntax is the same as `Swap1` and involves no addressing or dereferencing. The following `main` function illustrates the differences:

```
int main (void)
{
    int i = 10, j = 20;
    Swap1(i, j);    cout << i << ", " << j << '\n';
    Swap2(&i, &j);  cout << i << ", " << j << '\n';
    Swap3(i, j);        cout << i << ", " << j << '\n';
}
```

When run, it will produce the following output:

```
10, 20
20, 10
10, 20
```

□

Typedefs

Typedef is a syntactic facility for introducing symbolic names for data types. Just as a reference defines an alias for an object, a typedef defines an alias for a type. Its main use is to simplify otherwise complicated type declarations as an aid to improved readability. Here are a few examples:

```
typedef char *String;
Typedef char Name[12];
typedef unsigned int uint;
```

The effect of these definitions is that `String` becomes an alias for `char*`, `Name` becomes an alias for an array of 12 `char`s, and `uint` becomes an alias for `unsigned int`. Therefore:

```
String  str;        // is the same as: char *str;
Name    name;       // is the same as: char name[12];
uint    n;          // is the same as: unsigned int n;
```

The complicated declaration of `Compare` in Listing 5.8 is a good candidate for typedef:

```
typedef int (*Compare)(const char*, const char*);

int BinSearch (char *item, char *table[], int n, Compare comp)
{
    //...
        if ((cmp = comp(item, table[mid])) == 0)
            return mid;
    //...
}
```

The typedef introduces `Compare` as a new type name for any function with the given prototype. This makes `BinSearch`'s signature arguably simpler.

□

Exercises

5.1 Define two functions which, respectively, input values for the elements of an array of reals and output the array elements:

```
void ReadArray  (double nums[], const int size);
void WriteArray (double nums[], const int size);
```

5.2 Define a function which reverses the order of the elements of an array of reals:

```
void Reverse (double nums[], const int size);
```

5.3 The following table specifies the major contents of four brands of breakfast cereals. Define a two-dimensional array to capture this data:

	Fiber	Sugar	Fat	Salt
Top Flake	12g	25g	16g	0.4g
Cornabix	22g	4g	8g	0.3g
Oatabix	28g	5g	9g	0.5g
Ultrabran	32g	7g	2g	0.2g

Write a function which outputs this table element by element.

5.4 Define a function to input a list of names and store them as dynamically-allocated strings in an array, and a function to output them:

```
void ReadNames (char *names[], const int size);
void WriteNames (char *names[], const int size);
```

Write another function which sorts the list using bubble sort:

```
void BubbleSort (char *names[], const int size);
```

Bubble sort involves repeated scans of the list, where during each scan adjacent items are compared and swapped if out of order. A scan which involves no swapping indicates that the list is sorted.

5.5 Rewrite the following function using pointer arithmetic:

```
char* ReverseString (char *str)
{
    int len = strlen(str);
    char *result = new char[len + 1];

    for (register i = 0; i < len; ++i)
        result[i] = str[len - i - 1];
    result[len] = '\0';
    return result;
}
```


5.6 Rewrite BubbleSort (from 5.4) so that it uses a function pointer for comparison of names.

5.7 Rewrite the following using typedefs:

```
void (*Swap)(double, double);
char *table[];
char *&name;
usigned long *values[10][20];
```

□

6. Classes

This chapter introduces the class construct of C++ for defining new data types. A data type consists of two things:

- A concrete representation of the objects of the type.
- A set of operations for manipulating the objects.

Added to these is the restriction that, other than the designated operations, no other operation should be able to manipulate the objects. For this reason, we often say that the operations *characterize* the type, that is, they decide what can and what cannot happen to the objects. For the same reason, proper data types as such are often called **abstract data types** – abstract because the internal representation of the objects is hidden from operations that do not belong to the type.

A **class definition** consists of two parts: header and body. The class **header** specifies the class **name** and its **base classes**. (The latter relates to derived classes and is discussed in Chapter 8.) The class **body** defines the class **members**. Two types of members are supported:

- **Data members** have the syntax of variable definitions and specify the representation of class objects.
- **Member functions** have the syntax of function prototypes and specify the class operations, also called the class **interface**.

Class members fall under one of three different access permission categories:

- **Public** members are accessible by all class users.
- **Private** members are only accessible by the class members.
- **Protected** members are only accessible by the class members and the members of a derived class.

The data type defined by a class is used in exactly the same way as a built-in type.

A Simple Class

Listing 6.1 shows the definition of a simple class for representing points in two dimensions.

Listing 6.1

```
1  class Point {
2      int  xVal, yVal;
3  public:
4      void SetPt (int, int);
5      void OffsetPt (int, int);
6  };
```

Annotation

1 This line contains the class header and names the class as `Point`. A class definition always begins with the keyword `class`, followed by the class name. An open brace marks the beginning of the class body.

2 This line defines two data members, `xVal` and `yVal`, both of type `int`. The default access permission for a class member is private. Both `xVal` and `yVal` are therefore private.

3 This keyword specifies that from this point onward the class members are public.

4-5 These two are public member functions. Both have two integer parameters and a void return type.

6 This brace marks the end of the class body.

The order in which the data and member functions of a class are presented is largely irrelevant. The above class, for example, may be equivalently written as:

```
class Point {
public:
    void SetPt (int, int);
    void OffsetPt (int, int);
private:
    int  xVal, yVal;
};
```

The actual definition of the member functions is usually not part of the class and appears separately. Listing 6.2 shows the separate definition of `SetPt` and `OffsetPt`.

Listing 6.2

```
 1  void Point::SetPt (int x, int y)
 2  {
 3      xVal = x;
 4      yVal = y;
 5  }

 6  void Point::OffsetPt (int x, int y)
 7  {
 8      xVal += x;
 9      yVal += y;
10  }
```

Annotation

1 The definition of a class member function is very similar to a normal function. The function name should be preceded by the class name and a double-colon. This identifies `SetPt` as being a member of `Point`. The function interface must match its earlier interface definition within the class (i.e., take two integer parameters and have the return type void).

3-4 Note how `SetPt` (being a member of `Point`) is free to refer to `xVal` and `yVal`. Non-member functions do not have this privilege.

Once a class is defined in this way, its name denotes a new data type, allowing us to define variables of that type. For example:

```
Point pt;           // pt is an object of class Point
pt.SetPt(10,20);    // pt is set to (10,20)
pt.OffsetPt(2,2);   // pt becomes (12,22)
```

Member functions are called using the dot notation: `pt.SetPt(10,20)` calls `SetPt` for the object `pt`, that is, `pt` is an implicit argument to `SetPt`.

By making `xVal` and `yVal` private members of the class, we have ensured that a user of the class cannot manipulate them directly:

```
pt.xVal = 10;       // illegal
```

This will not compile.

At this stage, we should clearly distinguish between object and class. A class denotes a type, of which there is only one. An object is an element of a particular type (class), of which there may be many. For example,

```
Point pt1, pt2, pt3;
```

defines three objects (`pt1`, `pt2`, and `pt3`) all of the same class (`Point`). Furthermore, operations of a class are applied to objects of that class, but never the class itself. A class is therefore a concept that has no concrete existence other than that reflected by its objects. □

Inline Member Functions

Just as global functions may be defined to be inline, so can the member functions of a class. In the class `Point`, for example, both member functions are very short (only two statements). Defining these to be inline improves the efficiency considerably. A member function is defined to be inline by inserting the keyword `inline` before its definition.

```
inline void Point::SetPt (int x,int y)
{
    xVal = x;
    yVal = y;
}
```

An easier way of defining member functions to be inline is to include their definition *inside* the class.

```
class Point {
    int  xVal, yVal;
public:
    void SetPt (int x,int y)    { xVal = x; yVal = y; }
    void OffsetPt (int x,int y) { xVal += x; yVal += y; }
};
```

Note that because the function body is included, no semicolon is needed after the prototype. Furthermore, all function parameters must be named.

□

Example: A Set Class

A set is an unordered collection of objects with no repetitions. This example shows how a set may be defined as a class. For simplicity, we will restrict ourselves to sets of integers with a finite number of elements. Listing 6.3 shows the `Set` class definition.

Listing 6.3

```
 1  #include <iostream.h>

 2  const   maxCard = 100;
 3  enum    Bool {false, true};

 4  class Set {
 5  public:
 6      void    EmptySet    (void)      { card = 0; }
 7      Bool    Member      (const int);
 8      void    AddElem     (const int);
 9      void    RmvElem     (const int);
10      void    Copy            (Set&);
11      Bool    Equal       (Set&);
12      void    Intersect   (Set&, Set&);
13      void    Union       (Set&, Set&);
14      void    Print       (void);
15  private:
16      int     elems[maxCard];         // set elements
17      int     card;                   // set cardinality
18  };
```

Annotation

2 `MaxCard` denotes the maximum number of elements a set may have.

6 `EmptySet` clears the contents of the set by setting its cardinality to zero.

7 `Member` checks if a given number is an element of the set.

8 `AddElem` adds a new element to the set. If the element is already in the set then nothing happens. Otherwise, it is inserted. Should this result in an overflow then the element is not inserted.

9 `RmvElem` removes an existing element from the set, provided that element is already in the set.

10 `Copy` copies one set to another. The parameter of this function is a reference to the destination set.

11 `Equal` checks if two sets are equal. Two sets are equal if they contain exactly the same elements (the order of which is immaterial).

12 `Intersect` compares two sets to produce a third set (denoted by its last parameter) whose elements are in both sets. For example, the intersection of {2,5,3} and {7,5,2} is {2,5}.

13 `Union` compares two sets to produce a third set (denoted by its last parameter) whose elements are in either or both sets. For example, the union of {2,5,3} and {7,5,2} is {2,5,3,7}.

14 `Print` prints a set using the conventional mathematical notation. For example, a set containing the numbers 5, 2, and 10 is printed as {5,2,10}.

16 The elements of the set are represented by the `elems` array.

17 The cardinality of the set is denoted by `card`. Only the first `card` entries in `elems` are considered to be valid elements.

The separate definition of the member functions of a class is sometimes referred to as the **implementation** of the class. The implementation of the `Set` class is as follows.

```
Bool Set::Member (const int elem)
{
    for (register i = 0; i < card; ++i)
       if (elems[i] == elem)
           return true;
    return false;
}

void Set::AddElem (const int elem)
{
    if (Member(elem))
       return;
    if (card < maxCard)
       elems[card++] = elem;
    else
       cout << "Set overflow\n";
}

void Set::RmvElem (const int elem)
{
    for (register i = 0; i < card; ++i)
       if (elems[i] == elem) {
           for (; i < card-1; ++i) // shift elements left
               elems[i] = elems[i+1];
           --card;
       }
}


void Set::Copy (Set &set)
{
    for (register i = 0; i < card; ++i)
```

```
        set.elems[i] = elems[i];
    set.card = card;
}

Bool Set::Equal (Set &set)
{
    if (card != set.card)
        return false;
    for (register i = 0; i < card; ++i)
        if (!set.Member(elems[i]))
            return false;
    return true;
}

void Set::Intersect (Set &set, Set &res)
{
    res.card = 0;
    for (register i = 0; i < card; ++i)
        if (set.Member(elems[i]))
            res.elems[res.card++] = elems[i];
}

void Set::Union (Set &set, Set &res)
{
    set.Copy(res);
    for (register i = 0; i < card; ++i)
        res.AddElem(elems[i]);
}

void Set::Print (void)
{
    cout << "{";
    for (int i = 0; i < card-1; ++i)
        cout << elems[i] << ",";
    if (card > 0)      // no comma after the last element
        cout << elems[card-1];
    cout << "}\n";
}
```

The following `main` function creates three `Set` objects and exercises some of its member functions.

```
int main (void)
{
    Set    s1, s2, s3;

    s1.EmptySet();  s2.EmptySet();  s3.EmptySet();
    s1.AddElem(10); s1.AddElem(20); s1.AddElem(30); s1.AddElem(40);
    s2.AddElem(30); s2.AddElem(50); s2.AddElem(10); s2.AddElem(60);

    cout << "s1 = ";   s1.Print();
    cout << "s2 = ";   s2.Print();

    s2.RmvElem(50);         cout << "s2 - {50} = ";      s2.Print();
    if (s1.Member(20))  cout << "20 is in s1\n";
```



```
    s1.Intersect(s2,s3);   cout << "s1 intsec s2 = ";  s3.Print();
    s1.Union(s2,s3);       cout << "s1 union s2 = ";   s3.Print();
    if (!s1.Equal(s2))  cout << "s1 /= s2\n";
    return 0;
}
```

When run, the program will produce the following output:

```
s1 = {10,20,30,40}
s2 = {30,50,10,60}
s2 - {50} = {30,10,60}
20 is in s1
s1 intsec s2 = {10,30}
s1 union s2 = {30,10,60,20,40}
s1 /= s2
```

□

Constructors

It is possible to define and at the same time initialize objects of a class. This is supported by special member functions called constructors. A constructor always has the same name as the class itself. It never has an explicit return type. For example,

```
class Point {
    int  xVal, yVal;
public:
         Point (int x,int y) {xVal = x; yVal = y;} // constructor
    void OffsetPt (int,int);
};
```

is an alternative definition of the `Point` class, where `SetPt` has been replaced by a constructor, which in turn is defined to be inline.

Now we can define objects of type `Point` and initialize them at once. This is in fact compulsory for classes that contain constructors that require arguments:

```
Point pt1 = Point(10,20);
Point pt2;                  // illegal!
```

The former can also be specified in an abbreviated form.

```
Point pt1(10,20);
```

A class may have more than one constructor. To avoid ambiguity, however, each of these must have a unique signature. For example,

```
class Point {
    int  xVal, yVal;
public:
         Point (int x, int y)   { xVal = x;  yVal = y; }
         Point (float, float);              // polar coordinates
         Point (void)           { xVal = yVal = 0; }  // origin
    void OffsetPt (int, int);
};

Point::Point (float len, float angle)   // polar coordinates
{
    xVal = (int) (len * cos(angle));
    yVal = (int) (len * sin(angle));
}
```

offers three different constructors. An object of type `Point` can be defined using any of these:

```
Point pt1(10,20);       // cartesian coordinates
Point pt2(60.3,3.14);   // polar coordinates
Point pt3;              // origin
```


The `Set` class can be improved by using a constructor instead of `EmptySet`:

```
class Set {
public:
    Set (void)      { card = 0; }
    //...
};
```

This has the distinct advantage that the programmer need no longer remember to call `EmptySet`. The constructor ensures that every set is initially empty.

The `Set` class can be further improved by giving the user control over the maximum size of a set. To do this, we define `elems` as an integer pointer rather than an integer array. The constructor can then be given an argument which specifies the desired size. This means that `maxCard` will no longer be the same for all `Set` objects and therfore needs to become a data member itself:

```
class Set {
public:
            Set (const int size);
    //...
private:
    int     *elems;             // set elements
    int     maxCard;                // maximum cardinality
    int     card;               // set cardinality
};
```

The constructor simply allocates a dynamic array of the desired size and initializes `maxCard` and `card` accordingly:

```
Set::Set (const int size)
{
    elems = new int[size];
    maxCard = size;
    card = 0;
}
```

It is now possible to define sets of different maximum sizes:

```
Set ages(10), heights(20), primes(100);
```

It is important to note that an object's constructor is applied when the object is *created*. This in turn depends on the object's scope. For example, a global object is created as soon as program execution commences; an automatic object is created when its scope is entered; and a dynamic object is created when the `new` operator is applied to it. □

Destructors

Just as a constructor is used to initialize an object when it is created, a destructor is used to clean up the object just before it is destroyed. A destructor always has the same name as the class itself, but is preceded with a ~ symbol. Unlike constructors, a class may have at most one destructor. A destructor never takes any arguments and has no explicit return type.

Destructors are generally useful for classes which have pointer data members which point to memory blocks allocated by the class itself. In such cases it is important to release member-allocated memory before the object is destroyed. A destructor can do just that.

For example, our revised version of `Set` uses a dynamically-allocated array for the `elems` member. This memory should be released by a destructor:

```
class Set {
public:
            Set         (const int size);
            ~Set        (void)  {delete elems;} // destructor
    //...
private:
    int     *elems;     // set elements
    int     maxCard;        // maximum cardinality
    int     card;       // set cardinality
};
```

Now consider what happens when a `Set` is defined and used in a function:

```
void Foo (void)
{
    Set  s(10);
    //...
}
```

When `Foo` is called, the constructor for `s` is invoked, allocating storage for `s.elems` and initializing its data members. Next the rest of the body of `Foo` is executed. Finally, before `Foo` returns, the destructor for `s` is invoked, deleting the storage occupied by `s.elems`. Hence, as far as storage allocation is concerned, `s` behaves just like an automatic variable of a built-in type, which is created when its scope is entered and destroyed when its scope is left.

In general, an object's constructor is applied just before the object is *destroyed*. This in turn depends on the object's scope. For example, a global object is destroyed when program execution is completed; an automatic object is destroyed when its scope is left; and a dynamic object is destroyed when the `delete` operator is applied to it. □

Friends

Occasionally we may need to grant a function access to the nonpublic members of a class. Such an access is obtained by declaring the function a friend of the class. There are two possible reasons for requiring this access:

- It may be the only correct way of defining the function.
- It may be necessary if the function is to be implemented efficiently.

Examples of the first case will be provided in Chapter 7, when we discuss overloaded input/output operators. An example of the second case is discussed below.

Suppose that we have defined two variants of the `Set` class, one for sets of integers and one for sets of reals:

```
class IntSet {
public:
    //...
private:
    int elems[maxCard];
    int card;
};

class RealSet {
public:
    //...
private:
    float elems[maxCard];
    int   card;
};
```

We want to define a function, `SetToReal`, which converts an integer set to a real set. We can do this by making the function a member of `IntSet`:

```
void IntSet::SetToReal (RealSet &set)
{
    set.EmptySet();
    for (register i = 0; i < card; ++i)
        set.AddElem((float) elems[i]);
}
```

Although this works, the overhead of calling `AddElem` for every member of the set may be unacceptable. The implementation can be improved if we could gain access to the private members of both `IntSet` and `RealSet`. This can be arranged by declaring `SetToReal` as a friend of `RealSet`.

```
class RealSet {
    //...
    friend void IntSet::SetToReal (RealSet&);
};
```

```
void IntSet::SetToReal (RealSet &set)
{
    set.card = card;
    for (register i = 0; i < card; ++i)
        set.elems[i] = (float) elems[i];
}
```

The extreme case of having all member functions of a class `A` as friends of another class `B` can be expressed in an *abbreviated* form:

```
class A;
class B {
    //...
    friend class A;     // abbreviated form
};
```

Another way of implementing `SetToReal` is to define it as a global function which is a friend of both classes:

```
class IntSet {
    //...
    friend void SetToReal (IntSet&, RealSet&);
};

class RealSet {
    //...
    friend void SetToReal (IntSet&, RealSet&);
};

void SetToReal (IntSet &iSet, RealSet &rSet)
{
    rSet.card = iSet.card;
    for (int i = 0; i < iSet.card; ++i)
        rSet.elems[i] = (float) iSet.elems[i];
}
```

Although a friend declaration appears inside a class, that does *not* make the function a member of that class. In general, the position of a friend declaration in a class is irrelevant: whether it appears in the private, protected, or the public section, it has the same meaning.

□

Default Arguments

As with global functions, a member function of a class may have default arguments. The same rules apply: all default arguments should be trailing arguments, and the argument should be an expression consisting of objects defined within the scope in which the class appears.

For example, a constructor for the `Point` class may use default arguments to provide more variations of the way a `Point` object may be defined:

```
class Point {
    int  xVal, yVal;
public:
         Point (int x = 0, int y = 0);
    //...
};
```

Given this constructor, the following definitions are all valid:

```
Point   p1;             // same as: p1(0, 0)
Point   p2(10);         // same as: p2(10, 0)
Point   p3(10, 20);
```

Careless use of default arguments can lead to undesirable ambiguity. For example, given the class

```
class Point {
    int  xVal, yVal;
public:
         Point (int x = 0, int y = 0);
         Point (float x = 0, float y = 0);  // polar coordinates
    //...
};
```

the following definition will be rejected as ambiguous, because it matches both constructors:

```
Point p;                    // ambiguous!
```

□

Implicit Member Argument

When a class member function is called, it receives an implicit argument which denotes the particular object (of the class) for which the function is invoked. For example, in

```
Point pt(10,20);
pt.OffsetPt(2,2);
```

`pt` is an implicit argument to `OffsetPt`. Within the body of the member function, one can refer to this implicit argument explicitly as `this`, which denotes a pointer to the object for which the member is invoked. Using `this`, `OffsetPt` can be rewritten as:

```
Point::OffsetPt (int x, int y)
{
    this->xVal += x;        // equivalent to: xVal += x;
    this->yVal += y;        // equivalent to: yVal += y;
}
```

Use of `this` in this particular example is redundant. There are, however, programming cases where the use of the `this` pointer is essential. We will see examples of such cases in Chapter 7, when discussing overloaded operators.

The `this` pointer can be used for referring to member functions in exactly the same way as it is used for data members. It is important to bear in mind, however, that `this` is defined for use within member functions of a class only. In particular, it is undefined for global functions (including global friend functions).

□

Scope Operator

When calling a member function, we usually use an abbreviated syntax. For example:

```
pt.OffsetPt(2,2);           // abbreviated form
```

This is equivalent to the full form:

```
pt.Point::OffsetPt(2,2);    // full form
```

The full form uses the binary scope operator `::` to indicate that `OffsetPt` is a member of `Point`.

In some situations, using the scope operator is essential. For example, the case where the name of a class member is hidden by a local variable (e.g., member function parameter) can be overcome using the scope operator:

```
class Point {
public:
    Point (int x, int y)    { Point::x = x; Point::y = y; }
    //...
private:
    int x, y;
}
```

Here `x` and `y` in the constructor (inner scope) hide `x` and `y` in the class (outer scope). The latter are referred to explicitly as `Point::x` and `Point::y`.

□

Member Initialization List

There are two ways of initializing the data members of a class. The first approach involves initializing the data members using assignments in the body of a constructor. For example:

```
class Image {
public:
        Image   (const int w, const int h);
private:
    int width;
    int height;
    //...
};

Image::Image (const int w, const int h)
{
    width = w;
    height = h;
    //...
}
```

The second approach uses a **member initialization list** in the definition of a constructor. For example:

```
class Image {
public:
        Image   (const int w, const int h);
private:
    int width;
    int height;
    //...
};

Image::Image (const int w, const int h) : width(w), height(h)
{
    //...
}
```

The effect of this declaration is that `width` is initialized to `w` and `height` is initialized to `h`. The only difference between this approach and the previous one is that here members are initialized *before* the body of the constructor is executed.

A member initialization list may be used for initializing any data member of a class. It is always placed between the constructor header and body. A colon is used to separate it from the header. It should consist of a comma-separated list of data members whose initial value appears within a pair of brackets.

□

Constant Members

A class data member may defined as constant. For example:

```
class Image {
    const int   width;
    const int   height;
    //...
};
```

However, data member constants cannot be initialized using the same syntax as for other constants:

```
class Image {
    const int   width = 256;        // illegal initializer!
    const int   height = 168;   // illegal initializer!
    //...
};
```

The correct way to initialize a data member constant is through a member initialization list:

```
class Image {
public:
                Image   (const int w, const int h);
private:
    const int   width;
    const int   height;
    //...
};

Image::Image (const int w, const int h) : width(w), height(h)
{
    //...
}
```

As one would expect, no member function is allowed to assign to a constant data member.

A constant data member is not appropriate for defining the dimension of an array data member. For example, in

```
class Set {
public:
            Set         (void) : maxCard(10)        { card = 0; }
    //...
private:
    const   maxCard;
    int     elems[maxCard];     // illegal!
    int     card;
};
```

the array `elems` will be rejected by the compiler for not having a constant dimension. The reason for this being that `maxCard` is not bound to a value during compilation, but when the program is run and the constructor is invoked.

Member functions may also be defined as constant. This is used to specify which member functions of a class may be invoked for a constant object. For example,

```
class Set {
public:
            Set         (void)              { card = 0; }
    Bool    Member      (const int) const;
    void    AddElem     (const int);
    //...
};

Bool Set::Member (const int elem) const
{
    //...
}
```

defines `Member` as a constant member function. To do so, the keyword `const` is inserted after the function header, both inside the class and in the function definition.

A constant object can only be modified by the constant member functions of the class:

```
const Set s;
s.AddElem(10);      // illegal: AddElem not a const member
s.Member(10);       // ok
```

Given that a constant member function is allowed to be invoked for constant objects, it would be illegal for it to attempt to modify any of the class data members.

Constructors and destructors need never be defined as constant members, since they have permission to operate on constant objects. They are also exempted from the above rule and can assign to a data member of a constant object, unless the data member is itself a constant.

□

Static Members

A data member of a class can be defined to be static. This ensures that there will be exactly one copy of the member, shared by all objects of the class. For example, consider a `Window` class which represents windows on a bitmap display:

```
class Window {
    static Window   *first;     // linked-list of all windows
    Window          *next;      // pointer to next window
    //...
};
```

Here, no matter how many objects of type `Window` are defined, there will be only one instance of `first`. Like other static variables, a static data member is by default initialized to 0. It can be initialized to an arbitrary value in the same scope where the member function definitions appear:

```
Window *Window::first = &myWindow;
```

The alternative is to make such variables global, but this is exactly what static members are intended to avoid; by including the variable in a class, we can ensure that it will be inaccessible to anything outside the class.

Member functions can also be defined to be static. Semantically, a static member function is like a global function which is a friend of the class, but inaccessible outside the class. It does not receive an implicit argument and hence cannot refer to `this`. Static member functions are useful for defining call-back routines whose parameter lists are predetermined and outside the control of the programmer.

For example, the `Window` class might use a call-back function for repainting exposed areas of the window:

```
class Window {
    //...
    static void PaintProc (Event *event);    // call-back
};
```

Because static members are shared and do not rely on the `this` pointer, they are best referred to using the *class::member* syntax. For example, `first` and `PaintProc` would be referred to as `Window::first` and `Window::PaintProc`. Public static members can be referred to using this syntax by nonmember functions (e.g., global functions).

□

Member Pointers

Recall how a function pointer was used in Chapter 5 to pass the address of a comparison function to a search function. It is possible to obtain and manipulate the address of a member function of a class in a similar fashion. As before, the idea is to make a function more flexible by making it independent of another function.

The syntax for defining a pointer to a member function is slightly more complicated, since the class name *must* also be included in the function pointer type. For example,

```
typedef int (Table::*Compare)(const char*, const char*);
```

defines a member function pointer type called `Compare` for a class called `Table`. This type will match the address of any member function of `Table` which takes two constant character pointers and returns an `int`. `Compare` may be used for passing a pointer to a `Search` member of `Table`:

```
class Table {
public:
            Table   (const int slots);
    int     Search  (char *item, Compare comp);

    int     CaseSesitiveComp (const char*, const char*);
    int     NormalizedComp   (const char*, const char*);
private:
    int     slots;
    char    **entries;
};
```

The definition of `Table` includes two sample comparison member functions which can be passed to `Search`. `Search` has to use a slightly complicated syntax for invoking the comparison function via `comp`:

```
int Table::Search (char *item, Compare comp)
{
    int bot = 0;
    int top = slots - 1;
    int mid, cmp;

    while (bot <= top) {
        mid = (bot + top) / 2;
        if ((cmp = (this->*comp)(item, entries[mid])) == 0)
            return mid;          // return item index
        else if (cmp < 0)
            top = mid - 1;       // restrict search to lower half
        else
            bot = mid + 1;       // restrict search to upper half
    }
    return -1;                   // not found
}
```


Note that `comp` can only be invoked via a `Table` object (the `this` pointer is used in this case). None of the following attempts, though seemingly reasonable, will work:

```
(*comp)(item, entries[mid]);         // illegal: no class object!
(Table::*comp)(item, entries[mid]);  // illegal: no class object!
this->*comp(item, entries[mid]);     // illegal: need brackets!
```

The last attempt will be interpreted as:

```
this->*(comp(item, entries[mid]));   // unintended precedence!
```

Therefore the brackets around `this->*comp` are necessary. Using a `Table` object instead of `this` will require the following syntax:

```
Table tab(10);
(tab.*comp)(item, entries[mid])
```

`Search` can be called and passed either of the two comparison member functions of `Table`. For example:

```
tab.Search("Sydney", Table::NormalizedComp);
```

The address of a data member can be obtained using the same syntax as for a member function. For example,

```
int Table::*n = &Table::slots;
int m = this->*n;
int p = tab.*n;
```

The above class member pointer syntax applies to all members except for static. Static members are essentially global entities whose scope has been limited to a class. Pointers to static members use the conventional syntax of global entities.

In general, the same protection rules apply as before: to take the address of a class member (data or function) one should have access to it. For example, a function which does not have access to the private members of a class cannot take the address of any of those members.

□

References Members

A class data member may defined as reference. For example:

```
class Image {
    int width;
    int height;
    int &widthRef;
    //...
};
```

As with data member constants, a data member reference cannot be initialized using the same syntax as for other references:

```
class Image {
    int width;
    int height;
    int &widthRef = width;       // illegal!
    //...
};
```

The correct way to initialize a data member reference is through a member initialization list:

```
class Image {
public:
                Image   (const int w, const int h);
private:
    int width;
    int height;
    int &widthRef;
    //...
};

Image::Image (const int w, const int h) : widthRef(width)
{
    //...
}
```

This causes `widthRef` to be a reference for `width`.

□

Class Object Members

A data member of a class may be of a user-defined type, that is, an object of another class. For example, a `Rectangle` class may be defined using two `Point` data members which represent the top-left and bottom-right corners of the rectangle:

```
class Rectangle {
public:
            Rectangle (int left, int top, int right, int bottom);
    //...
private:
    Point   topLeft;
    Point   botRight;
};
```

The constructor for `Rectangle` should also initialize the two object members of the class. Assuming that `Point` has a constructor, this is done by including `topLeft` and `botRight` in the member initialization list of the constructor for `Rectangle`:

```
Rectangle::Rectangle (int left, int top, int right, int bottom)
    : topLeft(left,top), botRight(right,bottom)
{
}
```

If the constructor for `Point` takes no parameters, or if it has default arguments for all of its parameters, then the above member initialization list may be omitted. Of course, the constructor is still implicitly called.

The order of initialization is always as follows. First, the constructor for `topLeft` is invoked, followed by the constructor for `botRight`, and finally the constructor for `Rectangle` itself. Object destruction always follows the opposite direction. First the destructor for `Rectangle` (if any) is invoked, followed by the destructor for `botRight`, and finally for `topLeft`. The reason that `topLeft` is initialized before `botRight` is not that it appears first in the member initialization list, but because it appears before `botRight` in the class itself. Therefore, defining the constructor as follows would *not* change the initialization (or destruction) order:

```
Rectangle::Rectangle (int left, int top, int right, int bottom)
    : botRight(right,bottom), topLeft(left,top)
{
}
```

□

Object Arrays

An array of a user-defined type is defined and used much in the same way as an array of a built-in type. For example, a pentagon can be defined as an array of 5 points:

```
Point pentagon[5];
```

This definition assumes that `Point` has an 'argument-less' constructor (i.e., one which can be invoked without arguments). The constructor is applied to each element of the array.

The array can also be initialized using a normal array initializer. Each entry in the initialization list would invoke the constructor with the desired arguments. When the initializer has less entries than the array dimension, the remaining elements are initialized by the argument-less constructor. For example,

```
Point pentagon[5] = {
    Point(10,20), Point(10,30), Point(20,30), Point(30,20)
};
```

initializes the first four elements of `pentagon` to explicit points, and the last element is initialized to (0,0).

When the constructor can be invoked with a single argument, it is sufficient to just specify the argument. For example,

```
Set sets[4] = {10, 20, 20, 30};
```

is an abbreviated version of:

```
Set sets[4] = {Set(10), Set(20), Set(20), Set(30)};
```

An array of objects can also be created dynamically using `new`:

```
Point *petagon = new Point[5];
```

When the array is finally deleted using `delete`, a pair of `[]` should be included:

```
delete [] pentagon;      // destroys all array elements
```

Unless the `[]` is included, `delete` will have no way of knowing that `pentagon` denotes an array of points and not just a single point. The destructor (if any) is applied to the elements of the array in reverse order before the array is deleted. Omitting the `[]` will cause the destructor to be applied to just the first element of the array:

```
delete pentagon;         // destroys only the first element!
```


Since the objects of a dynamic array cannot be explicitly initialized at the time of creation, the class must have an argument-less constructor to handle the implicit initialization. When this implicit initialization is insufficient, the programmer can explicitly reinitialize any of the elements later:

```
pentagon[0].Point(10, 20);
pentagon[1].Point(10, 30);
//...
```

Dynamic object arrays are useful in circumstances where we cannot predetermine the size of the array. For example, a general polygon class has no way of knowing in advance how many vertices a polygon may have:

```
class Polygon {
public:
    //...
private:
    Point   *vertices;      // the vertices
    int     nVertices;      // the number of vertices
};
```

□

Class Scope

A class introduces a **class scope** much in the same way a function (or block) introduces a local scope. All the class members belong to the class scope and thus hide entities with identical names in the enclosing scope. For example, in

```
int fork (void);        // system fork

class Process {
    int fork (void);
    //...
};
```

the member function `fork` hides the global system function `fork`. The former can refer to the latter using the unary scope operator:

```
int Process::fork (void)
{
    int pid = ::fork();     // use global system fork
    //...
}
```

A class itself may be defined at any one of three possible scopes:

- At the global scope. This leads to a **global class**, whereby it can be referred to by all other scopes. The great majority of C++ classes (including all the examples presented so far in this chapter) are defined at the global scope.
- At the class scope of another class. This leads to a **nested class**, where a class is contained by another class.
- At the local scope of a block or function. This leads to a **local class**, where the class is completely contained by a block or function.

A nested class is useful when a class is used only by one other class. For example,

```
class Rectangle {           // a nested class
public:
            Rectangle (int, int, int, int);
    //..
private:
    class Point {
    public:
            Point   (int, int);
    private:
        int x, y;
    };
    Point   topLeft, botRight;
};
```


defines `Point` as nested by `Rectangle`. The member functions of `Point` may be defined either inline inside the `Point` class or at the global scope. The latter would require further qualification of the member function names by preceding them with `Rectangle::`

```
Rectangle::Point::Point (int x, int y)
{
    //...
}
```

A nested class may still be accessed outside its enclosing class by fully qualifying the class name. The following, for example, would be valid at any scope (assuming that `Point` is made public within `Rectangle`):

```
Rectangle::Point pt(1,1);
```

A local class is useful when a class is used by only one function — be it a global function or a member function — or even just one block. For example,

```
void Render (Image &image)
{
    class ColorTable {
    public:
        ColorTable  (void)                  { /* ... */ }
        AddEntry    (int r, int g, int b)   { /* ... */ }
        //...
    };

    ColorTable colors;
    //...
}
```

defines `ColorTable` as a class local to `Render`.

Unlike a nested class, a local class is not accessible outside the scope within which it is defined. The following, therefore, would be illegal at the global scope:

```
ColorTable ct;      // undefined!
```

A local class must be completely defined inside the scope in which it appears. All of its functions members, therefore, need to be defined inline inside the class. This implies that a local scope is not suitable for defining anything but very simple classes.

□

A **structure** is a class all of whose members are by default public. (Remember that all of the members of a class are by default private.) Structures are defined using the same syntax as classes, except that the keyword `struct` is used instead of `class`. For example,

```
struct Point {
         Point      (int, int);
    void OffsetPt   (int, int);
    int  x, y;
};
```

is equivalent to:

```
class Point {
public:
         Point      (int, int);
    void OffsetPt   (int, int);
    int  x, y;
};
```

The `struct` construct originated in C, where it could only contain data members. It has been retained mainly for backward compatibility reasons. In C, a structure can have an initializer with a syntax similar to that of an array. C++ allows such initializers for structures and classes all of whose data members are public:

```
class Employee {
public:
    char    *name;
    int     age;
    double  salary;
};

Employee emp = {"Jack", 24, 38952.25};
```

The initializer consists of values which are assigned to the data members of the structure (or class) in the order they appear. This style of initialization is largely superseded by constructors. Furthermore, it cannot be used with a class that has a constructor.

A **union** is a class all of whose data members are mapped to the same address within its object (rather than sequentially as is the case in a class). The size of an object of a union is, therefore, the size of its largest data member.

The main use of unions is for situations where an object may assume values of different types, but only one at a time. For example, consider an interpreter for a simple programming language, called P, which supports a number of data types such as: integers, reals, strings, and lists. A value in this language may be defined to be of the type:


```
union Value {
    long    integer;
    double  real;
    char    *string;
    Pair    list;
    //...
};
```

where `Pair` is itself a user-defined type for creating lists:

```
class Pair {
    Value   *head;
    Value   *tail;
    //...
};
```

Assuming that a `long` is 4 bytes, a `double` 8 bytes, and a pointer 4 bytes, an object of type `Value` would be exactly 8 bytes, i.e., the same as the size of a `double` or a `Pair` object (the latter being equal to two pointers).

An object in P can be represented by the class,

```
class Object {
private:
    enum ObjType {intObj, realObj, strObj, listObj};
    ObjType type;       // object type
    Value   val;        // object value
    //...
};
```

where `type` provides a way of recording what type of value the object currently has. For example, when `type` is set to `strObj`, `val.string` is used for referring to its value.

Because of the unique way in which its data members are mapped to memory, a union may *not* have a static data member or a data member which requires a constructor.

Like a structure, all of the members of a union are by default public. The keywords `private`, `public`, and `protected` may be used inside a `struct` or a `union` in exactly the same way they are used inside a class for defining private, public, and protected members.

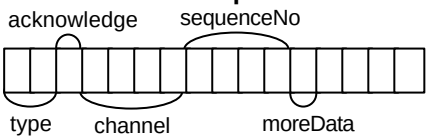
□

Bit Fields

It is sometimes desirable to directly control an object at the bit level, so that as many individual data items as possible can be packed into a bit stream without worrying about byte or word boundaries.

For example, in data communication, data is transferred in discrete units called packets. In addition to the user data that it carries, each packet also contains a header which is comprised of network-related information for managing the transmission of the packet across the network. To minimize the cost of transmission, it is desirable to minimize the space taken by the header. Figure 6.1 illustrates how the header fields are packed into adjacent bits to achieve this.

Figure 6.1 Header fields of a packet.



These fields can be expressed as **bit field** data members of a `Packet` class. A bit field may be defined to be of type `int` or `unsigned int`:

```
typedef unsigned int Bit;

class Packet {
    Bit type        : 2;    // 2 bits wide
    Bit acknowledge : 1;    // 1 bit wide
    Bit channel     : 4;    // 4 bits wide
    Bit sequenceNo  : 4;    // 4 bite wide
    Bit moreData    : 1;    // 1 bit wide
    //...
};
```

A bit field is referred to in exactly the same way as any other data member. Because a bit field does not necessarily start on a byte boundary, it is illegal to take its address. For the same reason, a bit field cannot be defined as static.

Use of enumerations can make working with bit fields easier. For example, given the enumerations

```
enum PacketType {dataPack, controlPack, supervisoryPack};
enum Bool       {false, true};
```

we can write:

```
Packet p;
p.type = controlPack;
p.acknowledge = true;
```

□

Exercises

6.1 Explain why the `Set` parameters of the `Set` member functions are declared as references.

6.2 Define a class named `Complex` for representing complex numbers. A complex number has the general form a + *i*b, where a is the real part and b is the imaginary part (*i* stands for imaginary). Complex arithmetic rules are as follows:

$$(a + ib) + (c + id) = (a + c) + i(b + d)$$
$$(a + ib) - (c + id) = (a + c) - i(b + d)$$
$$(a + ib) * (c + id) = (ac - bd) + i(bc + ad)$$

Define these operations as member functions of `Complex`.

6.3 Define a class named `Menu` which uses a linked-list of strings to represent a menu of options. Use a nested class, `Option`, to represent the set elements. Define a constructor, a destructor, and the following member functions for `Menu`:

- `Insert` which inserts a new option at a given position. Provide a default argument so that the item is appended to the end.
- `Delete` which deletes an existing option.
- `Choose` which displays the menu and invites the user to choose an option.

6.4 Redefine the `Set` class as a linked-list so that there would be no restriction on the number of elements a set may have. Use a nested class, `Element`, to represent the set elements.

6.5 Define a class named `Sequence` for storing sorted strings. Define a constructor, a destructor, and the following member functions for `Sequence`:

- `Insert` which inserts a new string into its sort position.
- `Delete` which deletes an existing string.
- `Find` which searches the sequence for a given string and returns true if it finds it, and false otherwise.
- `Print` which prints the sequence strings.

6.6 Define class named `BinTree` for storing sorted strings as a binary tree. Define the same set of member functions as for `Sequence` from the previous exercise.

6.7 Define a member function for `BinTree` which converts a sequence to a binary tree, as a friend of `Sequence`. Use this function to define a constructor for `BinTree` which takes a sequence as argument.

6.8 Add an integer ID data member to the `Menu` class (Exercise 6.3) so that all menu objects are sequentially numbered, starting from 0. Define an inline member function which returns the ID. How will you keep track of the last allocated ID?

6.9 Modify the `Menu` class so that an option can itself be a menu, thereby allowing nested menus.

□

7. Overloading

This chapter discusses the overloading of functions and operators in C++. The term *overloading* means 'providing multiple definitions of'. Overloading of functions involves defining distinct functions which share the same name, each of which has a unique signature. Function overloading is appropriate for:

- Defining functions which essentially do the same thing, but operate on different data types.
- Providing alternate interfaces to the same function.

Function overloading is purely a programming convenience.

Operators are similar to functions in that they take operands (arguments) and return a value. Most of the built-in C++ operators are already overloaded. For example, the + operator can be used to add two integers, two reals, or two addresses. Therefore, it has multiple definitions. The built-in definitions of the operators are restricted to built-in types. Additional definitions can be provided by the programmer, so that they can also operate on user-defined types. Each additional definition is implemented by a function.

The overloading of operators will be illustrated using a number of simple classes. We will discuss how type conversion rules can be used to reduce the need for multiple overloadings of the same operator. We will present examples of overloading a number of popular operators, including `<<` and `>>` for IO, `[]` and `()` for container classes, and the pointer operators. We will also discuss memberwise initialization and assignment, and the importance of their correct implementation in classes which use dynamically-allocated data members.

Unlike functions and operators, classes cannot be overloaded; each class must have a unique name. However, as we will see in Chapter 8, classes can be altered and extended through a facility called *inheritance*. Also functions and classes can be written as *templates*, so that they become independent of the data types they employ. We will discuss templates in Chapter 9.

Function Overloading

Consider a function, `GetTime`, which returns in its parameter(s) the current time of the day, and suppose that we require two variants of this function: one which returns the time as seconds from midnight, and one which returns the time as hours, minutes, and seconds. Given that these two functions serve the same purpose, there is no reason for them to have different names.

C++ allows functions to be overloaded, that is, the same function to have more than one definition:

```
long GetTime (void);         // seconds from midnight
void GetTime (int &hours, int &minutes, int &seconds);
```

When `GetTime` is called, the compiler compares the number and type of arguments in the call against the definitions of `GetTime` and chooses the one that matches the call. For example:

```
int h, m, s;
long t = GetTime();      // matches GetTime(void)
GetTime(h, m, s);        // matches GetTime(int&, int&, int&);
```

To avoid ambiguity, each definition of an overloaded function must have a unique signature.

Member functions of a class may also be overloaded:

```
class Time {
    //...
    long GetTime (void);         // seconds from midnight
    void GetTime (int &hours, int &minutes, int &seconds);
};
```

Function overloading is useful for obtaining flavors that are not possible using default arguments alone. Overloaded functions may also have default arguments:

```
void Error (int errCode, char *errMsg = "");
void Error (char *errMsg);
```

□

Operator Overloading

C++ allows the programmer to define additional meanings for its predefined operators by overloading them. For example, we can overload the `+` and `-` operators for adding and subtracting `Point` objects:

```
class Point {
public:
    Point (int x, int y)        {Point::x = x; Point::y = y;}
    Point operator + (Point &p) {return Point(x + p.x,y + p.y);}
    Point operator - (Point &p) {return Point(x - p.x,y - p.y);}
private:
    int x, y;
};
```

After this definition, `+` and `-` can be used for adding and subtracting points, much in the same way as they are used for adding and subtracting numbers:

```
Point p1(10,20), p2(10,20);
Point p3 = p1 + p2;
Point p4 = p1 - p2;
```

The above overloading of `+` and `-` uses member functions. Alternatively, an operator may be overloaded globally:

```
class Point {
public:
    Point (int x, int y)    {Point::x = x; Point::y = y;}
    friend Point operator + (Point &p, Point &q)
                            {return Point(p.x + q.x,p.y + q.y);}
    friend Point operator - (Point &p, Point &q)
                            {return Point(p.x - q.x,p.y - q.y);}
private:
    int x, y;
};
```

The use of an overloaded operator is equivalent to an explicit call to the function which implements it. For example:

```
operator+(p1, p2)       // is equivalent to: p1 + p2
```

In general, to overload a predefined operator λ, we define a function named `operator`λ . If λ is a binary operator:

- `operator`λ must take exactly one argument if defined as a class member, or two arguments if defined globally.

However, if λ is a unary operator:

- `operator`λ must take no arguments if defined as a member function, or one argument if defined globally.

Table 7.1 summarizes the C++ operators which can be overloaded. The remaining five operators cannot be overloaded:

```
.       .*      ::      ?:      sizeof      // not overloadable
```

Figure 7.1 Overloadable operators.

Unary:	+	-	*	!	~	&	++	--	()	->	->*
	new	delete									
Binary:	+	-	*	/	%	&	|	^	<<	>>	
	=	+=	-=	/=	%=	&=	|=	^=	<<=	>>=	
	==	!=	<	>	<=	>=	&&	||	[]	()	,

A strictly unary operator (e.g., ~) cannot be overloaded as binary, nor can a strictly binary operator (e.g., =) be overloaded as unary.

C++ does not support the definition of new operator tokens, because this can lead to ambiguity. Furthermore, the precedence rules for the predefined operators is fixed and cannot be altered. For example, no matter how you overload *, it will always have a higher precedence than +.

Operators ++ and -- can be overloaded as prefix as well as postfix. Equivalence rules do not hold for overloaded operators. For example, overloading + does not affect +=, unless the latter is also explicitly overloaded. Operators ->, =, [], and () can only be overloaded as member functions, and not globally.

To avoid the copying of large objects when passing them to an overloaded operator, references should be used. Pointers are not suitable for this purpose because an overloaded operator cannot operate exclusively on pointers.

□

Example: Set Operators

The `Set` class was introduced in Chapter 6. Most of the `Set` member functions are better defined as overloaded operators. Listing 7.1 illustrates.

Listing 7.1

```
#include <iostream.h>

const   maxCard = 100;
enum    Bool {false, true};

class Set {
public:
                Set         (void)      { card = 0; }
    friend Bool operator &  (const int, Set&);  // membership
    friend Bool operator == (Set&, Set&);       // equality
    friend Bool operator != (Set&, Set&);       // inequality
    friend Set  operator *  (Set&, Set&);       // intersection
    friend Set  operator +  (Set&, Set&);       // union
    //...
    void    AddElem (const int elem);
    void    Copy    (Set &set);
    void    Print   (void);
private:
    int     elems[maxCard];             // set elements
    int     card;                       // set cardinality
};
```

Here, we have decided to define the operator functions as global friends. They could have just as easily been defined as member functions. The implementation of these functions is as follow.

```
Bool operator & (const int elem, Set &set)
{
    for (register i = 0; i < set.card; ++i)
       if (elem == set.elems[i])
           return true;
    return false;
}

Bool operator == (Set &set1, Set &set2)
{
    if (set1.card != set2.card)
       return false;
    for (register i = 0; i < set1.card; ++i)
       if (!(set1.elems[i] & set2))        // use overloaded &
           return false;
    return true;
}

Bool operator != (Set &set1, Set &set2)
{
    return !(set1 == set2);               // use overloaded ==
}
```

```
Set operator * (Set &set1, Set &set2)
{
    Set res;

    for (register i = 0; i < set1.card; ++i)
       if (set1.elems[i] & set2)           // use overloaded &
           res.elems[res.card++] = set1.elems[i];
    return res;
}

Set operator + (Set &set1, Set &set2)
{
    Set res;

    set1.Copy(res);
    for (register i = 0; i < set2.card; ++i)
       res.AddElem(set2.elems[i]);
    return res;
}
```

The syntax for using these operators is arguably neater than those of the functions they replace, as illustrated by the following `main` function:

```
int main (void)
{
    Set    s1, s2, s3;

    s1.AddElem(10); s1.AddElem(20); s1.AddElem(30); s1.AddElem(40);
    s2.AddElem(30); s2.AddElem(50); s2.AddElem(10); s2.AddElem(60);

    cout << "s1 = ";   s1.Print();
    cout << "s2 = ";   s2.Print();

    if (20 & s1) cout << "20 is in s1\n";

    cout << "s1 intsec s2 = "; (s1 * s2).Print();
    cout << "s1 union s2 = ";  (s1 + s2).Print();

    if (s1 != s2) cout << "s1 /= s2\n";
    return 0;
}
```

When run, the program will produce the following output:

```
s1 = {10,20,30,40}
s2 = {30,50,10,60}
20 is in s1
s1 intsec s2 = {10,30}
s1 union s2 = {10,20,30,40,50,60}
s1 /= s2
```

□

Type Conversion

The normal built-in type conversion rules of the language also apply to overloaded functions and operators. For example, in

```
if ('a' & set)
    //...
```

the first operand of `&` (i.e., `'a'`) is implicitly converted from `char` to `int`, because overloaded `&` expects its first operand to be of type `int`.

Any other type conversion required in addition to these must be explicitly defined by the programmer. For example, suppose we want to overload `+` for the `Point` type so that it can be used to add two points, or to add an integer value to both coordinates of a point:

```
class Point {
    //...
    friend Point operator + (Point, Point);
    friend Point operator + (int, Point);
    friend Point operator + (Point, int);
};
```

To make `+` commutative, we have defined two functions for adding an integer to a point: one for when the integer is the first operand, and one for when the integer is the second operand. It should be obvious that if we start considering other types in addition to `int`, this approach will ultimately lead to an unmanageable variations of the operator.

A better approach is to use a constructor to convert the object to the same type as the class itself so that one overloaded operator can handle the job. In this case, we need a constructor which takes an `int`, specifying both coordinates of a point:

```
class Point {
    //...
    Point (int x)          { Point::x = Point::y = x; }
    friend Point operator + (Point, Point);
};
```

For constructors of one argument, one need not explicitly call the constructor:

```
Point p = 10;               // equivalent to: Point p(10);
```

Hence, it is possible to write expressions that involve variables or constants of type `Point` and `int` using the `+` operator.

```
Point p(10,20), q = 0;
q = p + 5;                  // equivalent to: q = p + Point(5);
```

Here, 5 is first converted to a temporary `Point` object and then added to `p`. The temporary object is then destroyed. The overall effect is an *implicit* type conversion from `int` to `Point`. The final value of `q` is therefore (15,25).

What if we want to do the opposite conversion, from the class type to another type? In this case, constructors cannot be used because they always return an object of the class to which they belong. Instead, one can define a member function which explicitly converts the object to the desired type.

For example, given a `Rectangle` class, we can define a type conversion function which converts a rectangle to a point, by overloading the type operator `Point` in `Rectangle`:

```
class Rectangle {
public:
    Rectangle (int left, int top, int right, int bottom);
    Rectangle (Point &p, Point &q);
    //...
    operator Point ()   {return botRight - topLeft;}

private:
    Point   topLeft;
    Point   botRight;
};
```

This operator is defined to convert a rectangle to a point, whose coordinates represent the width and height of the rectangle. Therefore, in the code fragment

```
Point       p(5,5);
Rectangle   r(10,10,20,30);
r + p;
```

rectangle `r` is first *implicitly* converted to a `Point` object by the type conversion operator, and then added to `p`.

The type conversion `Point` can also be applied *explicitly* using the normal type cast notation. For example:

```
Point(r);       // explicit type-cast to a Point
(Point)r;       // explicit type-cast to a Point
```

In general, given a user-defined type `X` and another (built-in or user-defined) type `Y`:

- A constructor defined for `X` which takes a single argument of type `Y` will implicitly convert `Y` objects to `X` objects when needed.
- Overloading the type operator `Y` in `X` will implicitly convert `X` objects to `Y` objects when needed.

```
class X {
    //...
    X (Y&);         // convert Y to X
```



```
    operator Y ();  // convert X to Y
};
```

One of the disadvantages of user-defined type conversion methods is that, unless they are used sparingly, they can lead to programs whose behaviors can be very difficult to predict. There is also the additional risk of creating ambiguity. Ambiguity occurs when the compiler has more than one option open to it for applying user-defined type conversion rules, and therefore unable to choose. All such cases are reported as errors by the compiler.

To illustrate possible ambiguities that can occur, suppose that we also define a type conversion constructor for `Rectangle` (which takes a `Point` argument) as well as overloading the `+` and `-` operators:

```
class Rectangle {
public:
    Rectangle (int left, int top, int right, int bottom);
    Rectangle (Point &p, Point &q);
    Rectangle (Point &p);

    operator Point ()   {return botRight - topLeft;}
    friend Rectangle operator + (Rectangle &r, Rectangle &t);
    friend Rectangle operator - (Rectangle &r, Rectangle &t);

private:
    Point   topLeft;
    Point   botRight;
};
```

Now, in

```
Point       p(5,5);
Rectangle   r(10,10,20,30);
r + p;
```

`r + p` can be interpreted in two ways. Either as

```
r + Rectangle(p)        // yields a Rectangle
```

or as:

```
Point(r) + p                // yields a Point
```

Unless the programmer resolves the ambiguity by explicit type conversion, this will be rejected by the compiler.

□

Example: Binary Number Class

Listing 7.2 defines a class for representing 16-bit binary numbers as sequences of 0 and 1 characters.

Listing 7.2

```
 1  #include <iostream.h>
 2  #include <string.h>

 3  int const binSize = 16;

 4  class Binary {
 5  public:
 6                  Binary       (const char*);
 7                  Binary       (unsigned int);
 8  friend  Binary  operator +   (const Binary, const Binary);
 9                  operator int ();    // type conversion
10          void    Print        (void);
11  private:
12          char    bits[binSize];          // binary quantity
13  };
```

Annotation

6 This constructor produces a binary number from its bit pattern.

7 This constructor converts a positive integer to its equivalent binary representation.

8 The `+` operator is overloaded for adding two binary numbers. Addition is done bit by bit. For simplicity, no attempt is made to detect overflows.

9 This type conversion operator is used to convert a `Binary` object to an `int` object.

10 This function simply prints the bit pattern of a binary number.

12 This array is used to hold the 0 and 1 bits of the 16-bit quantity as characters.

The implementation of these functions is as follows:

```
Binary::Binary (const char *num)
{
    int iSrc = strlen(num) - 1;
    int iDest = binSize - 1;

    while (iSrc >= 0 && iDest >= 0)     // copy bits
        bits[iDest--] = (num[iSrc--] == '0' ? '0' : '1');
    while (iDest >= 0)                  // pad left with zeros
        bits[iDest--] = '0';
}

Binary::Binary (unsigned int num)
{
```



```
        for (register i = binSize - 1; i >= 0; --i) {
            bits[i] = (num % 2 == 0 ? '0' : '1');
            num >>= 1;
        }
    }

    Binary operator + (const Binary n1, const Binary n2)
    {
        unsigned carry = 0;
        unsigned value;
        Binary res = "0";

        for (register i = binSize - 1; i >= 0; --i) {
            value = (n1.bits[i] == '0' ? 0 : 1) +
                    (n2.bits[i] == '0' ? 0 : 1) + carry;
            res.bits[i] = (value % 2 == 0 ? '0' : '1');
            carry = value >> 1;
        }
        return res;
    }

    Binary::operator int ()
    {
        unsigned value = 0;

        for (register i = 0; i < binSize; ++i)
            value = (value << 1) + (bits[i] == '0' ? 0 : 1);
        return value;
    }

    void Binary::Print (void)
    {
        char str[binSize + 1];
        strncpy(str, bits, binSize);
        str[binSize] = '\0';
        cout << str << '\n';
    }
```

The following `main` function creates two objects of type `Binary` and tests the `+` operator.

```
main ()
{
    Binary n1 = "01011";
    Binary n2 = "11010";
    n1.Print();
    n2.Print();
    (n1 + n2).Print();
    cout << n1 + Binary(5) << '\n';    // add and then convert to
int
    cout << n1 - 5 << '\n';    // convert n2 to int and then
subtract
}
```

The last two lines of `main` behave completely differently. The first of these converts 5 to `Binary`, does the addition, and then converts the `Binary` result to `int`, before sending it to `cout`. This is equivalent to:

```
cout << (int) Binary::operator+(n2,Binary(5)) << '\n';
```

The second converts `n1` to `int` (because `-` is not defined for `Binary`), performs the subtraction, and then send the result to `cout`. This is equivalent to:

```
cout << ((int) n2) - 5 << '\n';
```

In either case, the user-defined type conversion operator is applied implicitly. The output produced by the program is evidence that the conversions are performed correctly:

```
0000000000001011
0000000000011010
0000000000100101
16
6
```

□

Overloading << for Output

The simple and uniform treatment of output for built-in types is easily extended to user-defined types by further overloading the `<<` operator. For any given user-defined type `T`, we can define an `operator<<` function which outputs objects of type `T`:

```
ostream& operator << (ostream&, T&);
```

The first parameter must be a reference to `ostream` so that multiple uses of `<<` can be concatenated. The second parameter need not be a reference, but this is more efficient for large objects.

For example, instead of the `Binary` class's `Print` member function, we can overload the `<<` operator for the class. Because the first operand of `<<` must be an `ostream` object, it cannot be overloaded as a member function. It should therefore be defined as a global function:

```
class Binary {
    //...
    friend  ostream& operator << (ostream&, Binary&);
};

ostream& operator << (ostream &os, Binary &n)
{
    char str[binSize + 1];
    strncpy(str, n.bits, binSize);
    str[binSize] = '\0';
    cout << str;
    return os;
}
```

Given this definition, `<<` can be used for the output of binary numbers in a manner identical to its use for the built-in types. For example,

```
Binary n1 = "01011", n2 = "11010";
cout << n1 << " + " << n1 << " = " << n1 + n2 << '\n';
```

will produce the following output:

```
0000000000001011 + 0000000000011010 = 0000000000100101
```

In addition to its simplicity and elegance, this style of output eliminates the burden of remembering the name of the output function for each user-defined type. Without the use of overloaded `<<`, the last example would have to be written as (assuming that `\n` has been removed from `Print`):

```
Binary n1 = "01011", n2 = "11010";
n1.Print();     cout << " + ";    n2.Print();
cout << " = ";  (n1 + n2).Print(); cout << '\n';
```

□

Overloading >> for Input

Input of user-defined types is facilitated by overloading the `>>` operator, in a manner similar to the way `<<` is overloaded. For any given user-defined type `T`, we can define an `operator>>` function which inputs objects of type `T`:

```
istream& operator >> (istream&, T&);
```

The first parameter must be a reference to `istream` so that multiple uses of `>>` can be concatenated. The second parameter must be a reference, since it will be modified by the function.

Continuing with the `Binary` class example, we overload the `>>` operator for the input of bit streams. Again, because the first operand of `>>` must be an `istream` object, it cannot be overloaded as a member function:

```
class Binary {
    //...
    friend  istream& operator >> (istream&, Binary&);
};

istream& operator >> (istream &is, Binary &n)
{
    char str[binSize + 1];
    cin >> str;
    n = Binary(str);    // use the constructor for simplicity
    return is;
}
```

Given this definition, `>>` can be used for the input of binary numbers in a manner identical to its use for the built-in types. For example,

```
Binary n;
cin >> n;
```

will read a binary number from the keyboard into to `n`.

□

Overloading []

Listing 7.3 defines a simple associative vector class. An associative vector is a one-dimensional array in which elements can be looked up by their contents rather than their position in the array. In `AssocVec`, each element has a string name (via which it can be looked up) and an associated integer value.

Listing 7.3

```
 1  #include <iostream.h>
 2  #include <string.h>

 3  class AssocVec {
 4  public:
 5              AssocVec    (const int dim);
 6              ~AssocVec   (void);
 7      int&    operator [] (const char *idx);
 8  private:
 9      struct VecElem {
10          char    *index;
11          int     value;
12      }       *elems;         // vector elements
13      int     dim;            // vector dimension
14      int     used;           // elements used so far
15  };
```

Annotation

5 The constructor creates an associative vector of the dimension specified by its argument.

7 The overloaded `[]` operator is used for accessing vector elements. The function which overloads `[]` must have exactly one parameter. Given a string index, it searches the vector for a match. If a matching index is found then a reference to its associated value is returned. Otherwise, a new element is created and a reference to this value is returned.

12 The vector elements are represented by a dynamic array of `VecElem` structures. Each vector element consists of a string (denoted by `index`) and an integer value (denoted by `value`).

The implementation of the member functions is as follows:

```
AssocVec::AssocVec (const int dim)
{
    AssocVec::dim = dim;
    used = 0;
    elems = new VecElem[dim];
}

AssocVec::~AssocVec (void)
```

```
{
    for (register i = 0; i < used; ++i)
       delete elems[i].index;
    delete [] elems;
}

int& AssocVec::operator [] (const char *idx)
{
    for (register i = 0; i < used; ++i)    // search existing
elements
       if (strcmp(idx,elems[i].index) == 0)
           return elems[i].value;

    if (used < dim &&                  // create new element
        (elems[used].index = new char[strlen(idx)+1]) != 0) {
       strcpy(elems[used].index,idx);
       elems[used].value = used + 1;
       return elems[used++].value;
    }
    static int dummy = 0;
    return dummy;
}
```

Note that, because `AssocVec::operator[]` must return a valid reference, a reference to a dummy static integer is returned when the vector is full or when `new` fails.

A reference expression is an lvalue and hence can appear on both sides of an assignment. If a function returns a reference then a call to that function can be assigned to. This is why the return type of `AssocVec::operator[]` is defined to be a reference.

Using `AssocVec` we can now create associative vectors that behave very much like normal vectors:

```
AssocVec count(5);
count["apple"] = 5;
count["orange"] = 10;
count["fruit"] = count["apple"] + count["orange"];
```

This will set `count["fruit"]` to 15.

□

Overloading ()

Listing 7.4 defines a matrix class. A matrix is a table of values (very similar to a two-dimensional array) whose size is denoted by the number of rows and columns in the table. An example of a simple 2 x 3 matrix would be:

$$M = \begin{bmatrix} 10 & 20 & 30 \\ 21 & 52 & 19 \end{bmatrix}$$

The standard mathematical notation for referring to matrix elements uses brackets. For example, element 20 of *M* (i.e., in the first row and second column) is referred to as *M*(1,2). Matrix algebra provides a set of operations for manipulating matrices, which includes addition, subtraction, and multiplication.

Listing 7.4

```
 1  #include <iostream.h>

 2  class Matrix {
 3  public:
 4                  Matrix      (const short rows, const short cols);
 5                  ~Matrix     (void)             {delete elems;}
 6          double&  operator () (const short row, const short col);
 7  friend  ostream&     operator << (ostream&, Matrix&);
 8  friend  Matrix   operator  + (Matrix&, Matrix&);
 9  friend  Matrix   operator  - (Matrix&, Matrix&);
10  friend  Matrix   operator  * (Matrix&, Matrix&);

11  private:
12          const short rows;     // matrix rows
13          const short cols;     // matrix columns
14          double      *elems;   // matrix elements
15  };
```

Annotation

4 The constructor creates a matrix of the size specified by its arguments, all of whose elements are initialized to 0.

6 The overloaded `()` operator is used for accessing matrix elements. The function which overloads `()` may have zero or more parameters. It returns a reference to the specified element's value.

7 The overloaded `<<` is used for printing a matrix in tabular form.

8-10 These overloaded operators provide basic matrix operations.

14 The matrix elements are represented by a dynamic array of `double`s.

The implementation of the first three member functions is as follows:

```
Matrix::Matrix (const short r, const short c) : rows(r), cols(c)
{
    elems = new double[rows * cols];
}

double& Matrix::operator () (const short row, const short col)
{
    static  double dummy = 0.0;
    return (row >= 1 && row <= rows && col >= 1 && col <= cols)
           ? elems[(row - 1)*cols + (col - 1)]
           : dummy;
}

ostream& operator << (ostream &os, Matrix &m)
{
    for (register r = 1; r <= m.rows; ++r) {
       for (int c = 1; c <= m.cols; ++c)
           os << m(r,c) << " ";
       os << '\n';
    }
    return os;
}
```

As before, because `Matrix::operator()` must return a valid reference, a reference to a dummy static `double` is returned when the specified element does not exist. The following code fragment illustrates that matrix elements are lvalues:

```
Matrix m(2,3);
m(1,1) = 10;        m(1,2) = 20;        m(1,3) = 30;
m(2,1) = 15;        m(2,2) = 25;        m(2,3) = 35;
cout << m << '\n';
```

This will produce the following output:

```
10  20  30
15  25  35
```

□

Memberwise Initialization

Consider the following definition of the overloaded + operator for `Matrix`:

```
Matrix operator + (Matrix &p, Matrix &q)
{
    Matrix m(p.rows, p.cols);
    if (p.rows == q.rows && p.cols == q.cols)
        for (register r = 1; r <= p.rows; ++r)
            for (register c = 1; c <= p.cols; ++c)
                m(r,c) = p(r,c) + q(r,c);
    return m;
}
```

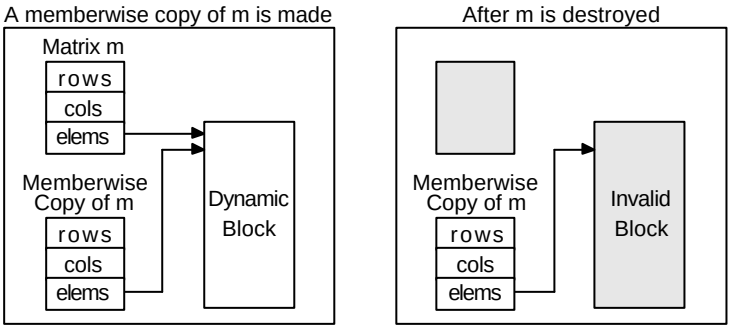
This function returns a matrix object which is initialized to `m`. The initialization is handled by an internal constructor which the compiler automatically generates for `Matrix`:

```
Matrix::Matrix (const Matrix &m) : rows(m.rows), cols(m.cols)
{
    elems = m.elems;
}
```

This form of initialization is called **memberwise initialization** because the special constructor initializes the object member by member. If the data members of the object being initialized are themselves objects of another class, then those are also memberwise initialized, etc.

As a result of the default memberwise initialization, the `elems` data member of both objects will point to the same dynamically-allocated block. However, `m` is destroyed upon the function returning. Hence the destructor deletes the block pointed to by `m.elems`, leaving the returned object's `elems` data member pointing to an invalid block! This ultimately leads to a runtime failure (typically a bus error). Figure 7.2 illustrates.

Figure 7.2 The danger of the default memberwise initialization of objects with pointers.



Memberwise initialization occurs in the following situations:

- When defining and initializing an object in a declaration statement that uses another object as its initializer, e.g., `Matrix n = m` in `Foo` below.
- When passing an object argument to a function (not applicable to a reference or pointer argument), e.g., `m` in `Foo` below.
- When returning an object value from a function (not applicable to a reference or pointer return value), e.g., `return n` in `Foo` below.

```
Matrix Foo (Matrix m)    // memberwise copy argument to m
{
    Matrix n = m;        // memberwise copy m to n
    //...
    return n;            // memberwise copy n and return copy
}
```

It should be obvious that default memberwise initialization is generally adequate for classes which have no pointer data members (e.g., `Point`). The problems caused by the default memberwise initialization of other classes can be avoided by explicitly defining the constructor in charge of memberwise initialization. For any given class `X`, the constructor always has the form:

```
X::X (const X&);
```

For example, for the `Matrix` class, this may be defined as follows:

```
class Matrix {
    Matrix (const Matrix&);
    //...
};

Matrix::Matrix (const Matrix &m) : rows(m.rows), cols(m.cols)
{
    int n = rows * cols;
    elems = new double[n];              // same size
    for (register i = 0; i < n; ++i)    // copy elements
        elems[i] = m.elems[i];
}
```

□

Memberwise Assignment

Objects of the same class are assigned to one another by an internal overloading of the = operator which is automatically generated by the compiler. For example, to handle the assignment in

```
Matrix m(2,2), n(2,2);
//...
m = n;
```

the compiler automatically generates the following internal function:

```
Matrix& Matrix::operator = (const Matrix &m)
{
    rows = m.rows;
    cols = m.cols;
    elems = m.elems;
}
```

This is identical in its approach to memberwise initialization and is called **memberwise assignment**. It suffers from exactly the same problems, which in turn can be overcome by explicitly overloading the = operator. For example, for the `Matrix` class, the following overloading of = would be appropriate:

```
Matrix& Matrix::operator = (const Matrix &m)
{
    if (rows == m.rows && cols == m.cols) {     // must match
        int n = rows * cols;
        for (register i = 0; i < n; ++i)        // copy elements
            elems[i] = m.elems[i];
    }
    return *this;
}
```

In general, for any given class X, the = operator is overloaded by the following member of X:

```
X& X::operator = (X&)
```

Operator = can only be overloaded as a member, and not globally.

□

Overloading new and delete

Objects of different classes usually have different sizes and frequency of usage. As a result, they impose different memory requirements. Small objects, in particular, are not efficiently handled by the default versions of `new` and `delete`. Every block allocated by `new` carries some overhead used for housekeeping purposes. For large objects this is not significant, but for small objects the overhead may be even bigger than the block itself. In addition, having too many small blocks can severely slow down subsequent allocation and deallocation. The performance of a program that dynamically creates many small objects can be significantly improved by using a simpler memory management strategy for those objects.

The dynamic storage management operators `new` and `delete` can be overloaded for a class, in which case they override the global definition of these operators when used for objects of that class.

As an example, suppose we wish to overload `new` and `delete` for the `Point` class, so that `Point` objects are allocated from an array:

```
#include <stddef.h>
#include <iostream.h>

const int  maxPoints = 512;

class Point {
public:
    //...
    void* operator new      (size_t bytes);
    void  operator delete   (void *ptr, size_t bytes);
private:
    int xVal, yVal;

    static union Block {
        int   xy[2];
        Block *next;
    }           *blocks;       // points to our freestore
    static Block *freeList;    // free-list of linked blocks
    static int   used;         // blocks used so far
};
```

The type name `size_t` is defined in `stddef.h`. `New` should always return a `void*`. The parameter of `new` denotes the size of the block to be allocated (in bytes). The corresponding argument is always automatically passed by the compiler. The first parameter of `delete` denotes the block to be deleted. The second parameter is optional and denotes the size of the allocated block. The corresponding arguments are automatically passed by the compiler.

Since `blocks`, `freeList` and `used` are static they do not affect the size of a `Point` object (it is still two integers). These are initialized as follows:

```
Point::Block *Point::blocks = new Block[maxPoints];
```



```
Point::Block *Point::freeList = 0;
int          Point::used = 0;
```

`New` takes the next available block from `blocks` and returns its address. `Delete` frees a block by inserting it in front of the linked-list denoted by `freeList`. When `used` reaches `maxPoints`, `new` removes and returns the first block in the linked-list, but fails (returns 0) when the linked-list is empty:

```
void* Point::operator new (size_t bytes)
{
    Block *res = freeList;
    return used < maxPoints
               ? &(blocks[used++])
               : (res == 0  ? 0
                            : (freeList = freeList->next, res));
}

void Point::operator delete (void *ptr, size_t bytes)
{
    ((Block*) ptr)->next = freeList;
    freeList = (Block*) ptr;
}
```

`Point::operator new` and `Point::operator delete` are invoked *only* for `Point` objects. Calling `new` with any other type as argument will invoke the global definition of `new`, even if the call occurs inside a member function of `Point`. For example:

```
Point *pt = new Point(1,1);      // calls Point::operator new
char  *str = new char[10];       // calls ::operator new
delete pt;                       // calls Point::operator delete
delete str;                      // calls ::operator delete
```

Even when `new` and `delete` are overloaded for a class, global `new` and `delete` are used when creating and destroying object arrays:

```
Point *points = new Point[5];    // calls ::operator new
//...
delete [] points;                // calls ::operator delete
```

The functions which overload `new` and `delete` for a class are always assumed by the compiler to be static, which means that they will not have access to the `this` pointer and therefore the nonstatic class members. This is because when these operators are invoked for an object of the class, the object does not exist: `new` is invoked *before* the object is constructed, and `delete` is called *after* it has been destroyed.

□

Overloading ->, *, and &

It is possible to divert the flow of control to a user-defined function before a pointer to an object is dereferenced using `->` or `*`, or before the address of an object is obtained using `&`. This can be used to do some extra pointer processing, and is facilitated by overloading unary operators `->`, `*`, and `&`.

For classes that do not overload `->`, this operator is always binary: the left operand is a pointer to a class object and the right operand is a class member name. When the left operand of `->` is an object or reference of type `X` (but not pointer), `X` is expected to have overloaded `->` as unary. In this case, `->` is first applied to the left operand to produce a result `p`. If `p` is a pointer to a class `Y` then `p` is used as the left operand of binary `->` and the right operand is expected to be a member of `Y`. Otherwise, `p` is used as the left operand of unary `->` and the whole procedure is repeated for class `Y`. Consider the following classes that overload `->`:

```
class A {
    //...
    B&  operator -> (void);
};

class B {
    //...
    Point* operator -> (void);
};
```

The effect of applying `->` to an object of type `A`

```
A   obj;
int i = obj->xVal;
```

is the successive application of overloaded `->` in `A` and `B`:

```
int i = (B::operator->(A::operator->(obj)))->xVal;
```

In other words, `A::operator->` is applied to `obj` to give `p`, `B::operator->` is applied to `p` to give `q`, and since `q` is a pointer to `Point`, the final result is `q->xVal`.

Unary operators `*` and `&` can also be overloaded so that the semantic correspondence between `->`, `*`, and `&` is preserved.

As an example, consider a library system which represents a book record as a raw string of the following format:

```
"%Aauthor\0%Ttitle\0%Ppublisher\0%Ccity\0%Vvolume\0%Yyear\0\n"
```


Each field starts with a field specifier (e.g., %A specifies an author) and ends with a null character (i.e., \0). The fields can appear in any order. Also, some fields may be missing from a record, in which case a default value must be used.

For efficiency reasons we may want to keep the data in this format but use the following structure whenever we need to access the fields of a record:

```
struct Book {
    char    *raw;       // raw format (kept for reference)
    char    *author;
    char    *title;
    char    *publisher;
    char    *city;
    short   vol;
    short   year;
};
```

The default field values are denoted by a global Book variable:

```
Book defBook = {
    "raw", "Author?", "Title?", "Publisher?", "City?", 0, 0
};
```

We now define a class for representing raw records, and overload the unary pointer operators to map a raw record to a Book structure whenever necessary.

```
#include <iostream.h>
#include <stdlib.h>         // needed for atoi() below

int const cacheSize = 10;

class RawBook {
public:
            RawBook      (char *str)    { data = str; }
    Book*   operator -> (void);
    Book&   operator *  (void);
    Book*   operator &  (void);
private:
    Book*   RawToBook   (void);

    char    *data;
    static  Book  *cache;       // cache memory
    static  short curr;         // current record in cache
    static  short used;         // number of used cache records
};
```

To reduce the frequency of mappings from RawBook to Book, we have used a simple cache memory of 10 records. The corresponding static members are initialized as follows:

```
Book    *RawBook::cache = new Book[cacheSize];
short   RawBook::curr = 0;
short   RawBook::used = 0;
```

The private member function RawToBook searches the cache for a RawBook and returns a pointer to its corresponding Book structure. If the book is not in the cache, RawToBook loads the book at the current position in the cache:

```
Book* RawBook::RawToBook (void)
{
    char *str = data;
    for (register i = 0; i < used; ++i) // search cache
        if (data == cache[i].raw)
            return cache + i;
    curr = used < cacheSize ? used++    // update curr and used
                            : (curr < 9 ? ++curr : 0);
    Book *bk = cache + curr;            // the book
    *bk = defBook;                      // set default values
    bk->raw = data;

    for (;;) {
        while (*str++ != '%')           // skip to next specifier
            ;
        switch (*str++) {               // get a field
            case 'A': bk->author = str;     break;
            case 'T': bk->title  = str;     break;
            case 'P': bk->publisher = str;  break;
            case 'C': bk->city = str;       break;
            case 'V': bk->vol = atoi(str);  break;
            case 'Y': bk->year = atoi(str); break;
        }
        while (*str++ != '\0')          // skip till end of field
            ;
        if (*str == '\n') break;        // end of record
    }
    return bk;
}
```

The overloaded operators ->, *, and & are easily defined in terms of RawToBook:

```
Book* RawBook::operator -> (void)   {return RawToBook(); }
Book& RawBook::operator *  (void)   {return *RawToBook();}
Book* RawBook::operator &  (void)   {return RawToBook(); }
```

The identical definitions for -> and & should not be surprising since -> is unary in this context and semantically equivalent to &.

The following test case demonstrates that the operators behave as expected. It sets up two book records and prints each using different operators.

```
main ()
{
    RawBook r1("%AA. Peters\0%TBlue Earth\0%PPhedra\0%CSydney\0%
            Y1981\0\n");
    RawBook r2("%TPregnancy\0%AF. Jackson\0%Y1987\0%PMiles\0\n");
    cout << r1->author    << ", " << r1->title     << ", "
         << r1->publisher << ", " << r1->city      << ", "
```



```
        << (*r1).vol     << ", " << (*r1).year    << '\n';

    Book *bp = &r2;          // note use of &
    cout << bp->author    << ", " << bp->title     << ", "
         << bp->publisher << ", " << bp->city      << ", "
         << bp->vol       << ", " << bp->year      << '\n';
}
```

It will produce the following output:

```
A. Peters, Blue Earth, Phedra, Sydney, 0, 1981
F. Jackson, Pregnancy, Miles, City?, 0, 1987
```

□

Overloading ++ and --

The auto increment and auto decrement operators can be overloaded in both prefix and postfix form. To distinguish between the two, the postfix version is specified to take an extra integer argument. For example, the prefix and postfix versions of `++` may be overloaded for the `Binary` class as follows:

```
class Binary {
    //...
    friend  Binary  operator ++  (Binary&);         // prefix
    friend  Binary  operator ++  (Binary&, int);    // postfix
};
```

Although we have chosen to define these as global friend functions, they can also be defined as member functions. Both are easily defined in terms of the `+` operator defined earlier:

```
Binary operator ++ (Binary &n)          // prefix
{
    return n = n + Binary(1);
}

Binary operator ++ (Binary &n, int)     // postfix
{
    Binary m = n;
    n = n + Binary(1);
    return m;
}
```

Note that we have simply ignored the extra parameter of the postfix version. When this operator is used, the compiler automatically supplies a default argument for it.

The following code fragment exercises both versions of the operator:

```
Binary n1 = "01011";
Binary n2 = "11010";
cout << ++n1 << '\n';
cout << n2++ << '\n';
cout << n2   << '\n';
```

It will produce the following output:

```
0000000000001100
0000000000011010
0000000000011011
```

The prefix and postfix versions of `--` may be overloaded in exactly the same way.

□

Exercises

7.1 Write overloaded versions of a `Max` function which compares two integers, two reals, or two strings, and returns the 'larger' one.

7.2 Overload the following two operators for the `Set` class:

- Operator `-` which gives the difference of two sets (e.g. *s* `-` *t* gives a set consisting of those elements of *s* which are not in *t*).
- Operator `<=` which checks if a set is contained by another (e.g., *s* `<=` *t* is true if all the elements of *s* are also in *t*).

7.3 Overload the following two operators for the `Binary` class:

- Operator `-` which gives the difference of two binary values. For simplicity, assume that the first operand is always greater than the second operand.
- Operator `[]` which indexes a bit by its position and returns its value as a 0 or 1 integer.

7.4 Sparse matrices are used in a number of numerical methods (e.g., finite element analysis). A sparse matrix is one which has the great majority of its elements set to zero. In practice, sparse matrices of sizes up to 500×500 are not uncommon. On a machine which uses a 64-bit representation for reals, storing such a matrix as an array would require 2 megabytes of storage. A more economic representation would record only nonzero elements together with their positions in the matrix. Define a `SparseMatrix` class which uses a linked-list to record only nonzero elements, and overload the `+`, `-`, and `*` operators for it. Also define an appropriate memberwise initialization constructor and memberwise assignment operator for the class.

7.5 Complete the implementation of the following `String` class. Note that two versions of the constructor and `=` are required, one for initializing/assigning to a `String` using a `char*`, and one for memberwise initialization/assignment. Operator `[]` should index a string character using its position. Operator `+` should concatenate two strings.

```
class String {
public:
                String      (const char*);
                String      (const String&);
                String      (const short);
                ~String     (void);

    String&     operator =  (const char*);
    String&     operator =  (const String&);
```

```
    char&      operator [] (const short);
    int        Length      (void)      {return(len);}
friend String   operator + (const String&, const String&);
friend ostream& operator <<(ostream&, String&);

private:
    char       *chars;     // string characters
    short      len;        // length of string
};
```

7.6 A bit vector is a vector with binary elements, that is, each element is either a 0 or a 1. Small bit vectors are conveniently represented by unsigned integers. For example, an `unsigned char` can represent a bit vector of 8 elements. Larger bit vectors can be defined as arrays of such smaller bit vectors. Complete the implementation of the `Bitvec` class, as defined below. It should allow bit vectors of any size to be created and manipulated using the associated operators.

```
enum Bool {false, true};
typedef unsigned char uchar;

class BitVec {
public:
            BitVec         (const short dim);
            BitVec         (const char* bits);
            BitVec         (const BitVec&);
            ~BitVec        (void)          { delete vec; }
    BitVec& operator =     (const BitVec&);
    BitVec& operator &=    (const BitVec&);
    BitVec& operator |=    (const BitVec&);
    BitVec& operator ^=    (const BitVec&);
    BitVec& operator <<=   (const short);
    BitVec& operator >>=   (const short);
    int     operator []    (const short idx);
    void    Set            (const short idx);
    void    Reset          (const short idx);

    BitVec  operator ~     (void);
    BitVec  operator &     (const BitVec&);
    BitVec  operator |     (const BitVec&);
    BitVec  operator ^     (const BitVec&);
    BitVec  operator <<    (const short n);
    BitVec  operator >>    (const short n);
    Bool    operator ==    (const BitVec&);
    Bool    operator !=        (const BitVec&);
friend ostream& operator << (ostream&, BitVec&);

private:
    uchar  *vec;           // vector of 8*bytes bits
    short   bytes;         // bytes in the vector

};
```

□

8. Derived Classes

In practice, most classes are not entirely unique, but rather variations of existing ones. Consider, for example, a class named `RecFile` which represents a file of records, and another class named `SortedRecFile` which represents a *sorted* file of records. These two classes would have much in common. For example, they would have similar member functions such as `Insert`, `Delete`, and `Find`, as well as similar data members. In fact, `SortedRecFile` would be a specialized version of `RecFile` with the added property that its records are organized in sorted order. Most of the member functions in both classes would therefore be identical, while a few which depend on the fact that file is sorted would be different. For example, `Find` would be different in `SortedRecFile` because it can take advantage of the fact that the file is sorted to perform a binary search instead of the linear search performed by the `Find` member of `RecFile`.

Given the shared properties of these two classes, it would be tedious to have to define them independently. Clearly this would lead to considerable duplication of code. The code would not only take longer to write it would also be harder to maintain: a change to any of the shared properties would have to be consistently applied to both classes.

Object-oriented programming provides a facility called **inheritance** to address this problem. Under inheritance, a class can inherit the properties of an existing class. Inheritance makes it possible to define a variation of a class without redefining the new class from scratch. Shared properties are defined only once, and reused as often as desired.

In C++, inheritance is supported by **derived classes**. A derived class is like an ordinary class, except that its definition is based on one or more existing classes, called **base classes**. A derived class can share selected properties (function as well as data members) of its base classes, but makes no changes to the definition of any of its base classes. A derived class can itself be the base class of another derived class. The inheritance relationship between the classes of a program is called a **class hierarchy**.

A derived class is also called a **subclass**, because it becomes a subordinate of the base class in the hierarchy. Similarly, a base class may be called a **superclass**, because from it many other classes may be derived.

An illustrative Class

We will define two classes for the purpose of illustrating a number of programming concepts in later sections of this chapter. The two classes are defined in Listing 8.1 and support the creation of a directory of personal contacts.

Listing 8.1

```
 1  #include <iostream.h>
 2  #include <string.h>

 3  class Contact {
 4  public:
 5                  Contact     (const char *name,
 6                               const char *address, const char *tel);
 7                  ~Contact    (void);
 8      const char* Name        (void) const    {return name;}
 9      const char* Address     (void) const    {return address;}
10      const char* Tel         (void) const    {return tel;}
11  friend ostream& operator << (ostream&, Contact&);

12  private:
13      char        *name;      // contact name
14      char        *address;   // contact address
15      char        *tel;       // contact telephone number
16  };

17  //---------------------------------------------------------------------
18  class ContactDir {
19  public:
20                   ContactDir (const int maxSize);
21                   ~ContactDir(void);
22          void     Insert     (const Contact&);
23          void     Delete     (const char *name);
24          Contact*     Find       (const char *name);
25  friend  ostream&     operator <<(ostream&, ContactDir&);

26  private:
27          int      Lookup     (const char *name);

28          Contact **contacts; // list of contacts
29          int      dirSize;   // current directory size
30          int      maxSize;   // max directory size
31  };
```

Annotation

3 `Contact` captures the details (i.e., name, address, and telephone number) of a personal contact.

18 `ContactDir` allows us to insert into, delete from, and search a list of personal contacts.

22 `Insert` inserts a new contact into the directory. This will overwrite an existing contact (if any) with identical name.

23 `Delete` deletes a contact (if any) whose name matches a given name.

24 `Find` returns a pointer to a contact (if any) whose name matches a given name.

27 `Lookup` returns the slot index of a contact whose name matches a given name. If none exists then `Lookup` returns the index of the slot where such an entry should be inserted. `Lookup` is defined as private because it is an auxiliary function used only by `Insert`, `Delete`, and `Find`.

The implementation of the member function and friends is as follows:

```
Contact::Contact (const char *name,
                  const char *address, const char *tel)
{
    Contact::name = new char[strlen(name) + 1];
    Contact::address = new char[strlen(address) + 1];
    Contact::tel = new char[strlen(tel) + 1];
    strcpy(Contact::name, name);
    strcpy(Contact::address, address);
    strcpy(Contact::tel, tel);
}

Contact::~Contact (void)
{
    delete name;
    delete address;
    delete tel;
}

ostream    &operator << (ostream &os, Contact &c)
{
    os << "(" << c.name << " , "
       << c.address << " , " << c.tel << ")";
    return os;
}

ContactDir::ContactDir (const int max)
{
    typedef Contact *ContactPtr;
    dirSize = 0;
    maxSize = max;
    contacts = new ContactPtr[maxSize];
};

ContactDir::~ContactDir (void)
{
    for (register i = 0; i < dirSize; ++i)
       delete contacts[i];
    delete [] contacts;
}
```

```
void ContactDir::Insert (const Contact& c)
{
    if (dirSize < maxSize) {
       int idx = Lookup(c.Name());
       if (idx > 0 &&
           strcmp(c.Name(), contacts[idx]->Name()) == 0) {
           delete contacts[idx];
       } else {
           for (register i = dirSize; i > idx; --i) // shift right
               contacts[i] = contacts[i-1];
           ++dirSize;
       }
       contacts[idx] = new Contact(c.Name(), c.Address(), c.Tel());
    }
}

void ContactDir::Delete (const char *name)
{
    int idx = Lookup(name);
    if (idx < dirSize) {
       delete contacts[idx];
       --dirSize;
       for (register i = idx; i < dirSize; ++i)    // shift left
           contacts[i] = contacts[i+1];
    }
}

Contact *ContactDir::Find (const char *name)
{
    int idx = Lookup(name);
    return (idx < dirSize &&
           strcmp(contacts[idx]->Name(), name) == 0)
               ? contacts[idx]
               : 0;
}

int ContactDir::Lookup (const char *name)
{
    for (register i = 0; i < dirSize; ++i)
       if (strcmp(contacts[i]->Name(), name) == 0)
           return i;
    return dirSize;
}

ostream    &operator << (ostream &os, ContactDir &c)
{
    for (register i = 0; i < c.dirSize; ++i)
       os << *(c.contacts[i]) << '\n';
    return os;
}
```

The following main function exercises the `ContactDir` class by creating a small directory and calling the member functions:

```
int main (void)
```

```
{
    ContactDir  dir(10);
    dir.Insert(Contact("Mary", "11 South Rd", "282 1324"));
    dir.Insert(Contact("Peter", "9 Port Rd", "678 9862"));
    dir.Insert(Contact("Jane", "321 Yara Ln", "982 6252"));
    dir.Insert(Contact("Jack", "42 Wayne St", "663 2989"));
    dir.Insert(Contact("Fred", "2 High St", "458 2324"));

    cout << dir;
    cout << "Find Jane: " << *dir.Find("Jane") << '\n';
    dir.Delete("Jack");
    cout << "Deleted Jack\n";
    cout << dir;
    return 0;
};
```

When run, it will produce the following output:

```
(Mary , 11 South Rd , 282 1324)
(Peter , 9 Port Rd , 678 9862)
(Jane , 321 Yara Ln , 982 6252)
(Jack , 42 Wayne St , 663 2989)
(Fred , 2 High St , 458 2324)
Find Jane: (Jane , 321 Yara Ln , 982 6252)
Deleted Jack
(Mary , 11 South Rd , 282 1324)
(Peter , 9 Port Rd , 678 9862)
(Jane , 321 Yara Ln , 982 6252)
(Fred , 2 High St , 458 2324)
```

□

A Simple Derived Class

We would like to define a class called `SmartDir` which behaves the same as `ContactDir`, but also keeps track of the most recently looked-up entry. `SmartDir` is best defined as a derivation of `ContactDir`, as illustrated by Listing 8.2.

Listing 8.2

```
1  class SmartDir : public ContactDir {
2  public:
3              SmartDir(const int max) : ContactDir(max) {recent = 0;}
4      Contact* Recent  (void);
5      Contact* Find    (const char *name);

6  private:
7      char    *recent;    // the most recently looked-up name
8  };
```

Annotation

1 A derived class header includes the base classes from which it is derived. A colon separates the two. Here, `ContactDir` is specified to be the base class from which `SmartDir` is derived. The keyword `public` before `ContactDir` specifies that `ContactDir` is used as a public base class.

3 `SmartDir` has its own constructor which in turn invokes the base class constructor in its member initialization list.

4 `Recent` returns a pointer to the last looked-up contact (or 0 if there is none).

5 `Find` is redefined so that it can record the last looked-up entry.

7 This `recent` pointer is set to point to the name of the last looked-up entry.

The member functions are defined as follows:

```
Contact* SmartDir::Recent (void)
{
    return recent == 0 ? 0 : ContactDir::Find(recent);
}

Contact* SmartDir::Find (const char *name)
{
    Contact *c = ContactDir::Find(name);
    if (c != 0)
        recent = (char*) c->Name();
    return c;
}
```

Because `ContactDir` is a public base class of `SmartDir`, all the public members of `ContactDir` become public members of `SmartDir`. This means that we can invoke a member function such as `Insert` on a `SmartDir` object and this

will simply be a call to `ContactDir::Insert`. Similarly, all the private members of `ContactDir` become private members of `SmartDir`.

In accordance with the principles of information hiding, the private members of `ContactDir` will not be accessible by `SmartDir`. Therefore, `SmartDir` will be unable to access any of the data members of `ContactDir` as well as the private member function `Lookup`.

`SmartDir` redefines the `Find` member function. This should not be confused with overloading. There are two distinct definitions of this function: `ContactDir::Find` and `SmartDir::Find` (both of which have the same signature, though they can have different signatures if desired). Invoking `Find` on a `SmartDir` object causes the latter to be invoked. As illustrated by the definition of `Find` in `SmartDir`, the former can still be invoked using its full name.

The following code fragment illustrates that `SmartDir` behaves the same as `ContactDir`, but also keeps track of the most recently looked-up entry:

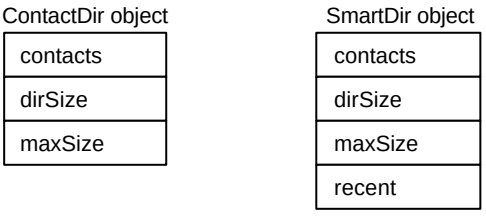
```
SmartDir        dir(10);
dir.Insert(Contact("Mary", "11 South Rd", "282 1324"));
dir.Insert(Contact("Peter", "9 Port Rd", "678 9862"));
dir.Insert(Contact("Jane", "321 Yara Ln", "982 6252"));
dir.Insert(Contact("Fred", "2 High St", "458 2324"));
dir.Find("Jane");
dir.Find("Peter");
cout << "Recent: " << *dir.Recent() << '\n';
```

This will produce the following output:

```
Recent: (Peter , 9 Port Rd , 678 9862)
```

An object of type `SmartDir` contains all the data members of `ContactDir` as well as any additional data members introduced by `SmartDir`. Figure 8.1 illustrates the physical make up of a `ContactDir` and a `SmartDir` object.

Figure 8.1 Base and derived class objects.



Class Hierarchy Notation

A class hierarchy is usually illustrated using a simple graph notation. Figure 8.2 illustrates the UML notation that we will be using in this book. Each class is represented by a box which is labeled with the class name. Inheritance between two classes is illustrated by a directed line drawn from the derived class to the base class. A line with a diamond shape at one end depicts **composition** (i.e., a class object is composed of one or more objects of another class). The number of objects contained by another object is depicted by a label (e.g., *n*).

Figure 8.2 A simple class hierarchy

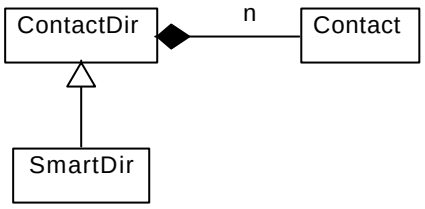


Figure 8.2 is interpreted as follows. `Contact`, `ContactDir`, and `SmartDir` are all classes. A `ContactDir` is composed of zero or more `Contact` objects. `SmartDir` is derived from `ContactDir`.

□

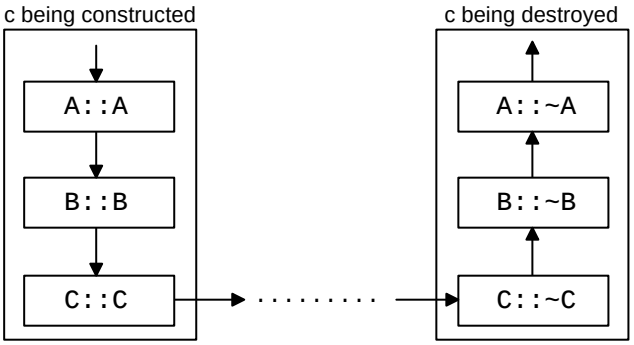
Constructors and Destructors

A derived class may have constructors and a destructor. Since a derived class may provide data members on top of those of its base class, the role of the constructor and destructor is to, respectively, initialize and destroy these additional members.

When an object of a derived class is created, the base class constructor is applied to it first, followed by the derived class constructor. When the object is destroyed, the destructor of the derived class is applied first, followed by the base class destructor. In other words, constructors are applied in order of derivation and destructors are applied in the reverse order. For example, consider a class `C` derived from `B` which is in turn derived from `A`. Figure 8.3 illustrates how an object `c` of type `C` is created and destroyed.

```
class A                 { /* ... */ }
class B : public A  { /* ... */ }
class C : public B  { /* ... */ }
```

Figure 8.3 Derived class object construction and destruction order.



The constructor of a derived class whose base class constructor requires arguments should specify these in the definition of its constructor. To do this, the derived class constructor explicitly invokes the base class constructor in its member initialization list. For example, the `SmartDir` constructor passes its argument to the `ContactDir` constructor in this way:

```
SmartDir::SmartDir (const int max) : ContactDir(max)
{ /* ... */ }
```

In general, all that a derived class constructor requires is an object from the base class. In some situations, this may not even require referring to the base class constructor:

```
extern ContactDir cd;           // defined elsewhere
SmartDir::SmartDir (const int max) : cd
{ /* ... */ }
```

□

Protected Class Members

Although the private members of a base class are inherited by a derived class, they are not accessible to it. For example, `SmartDir` inherits all the private (and public) members of `ContactDir`, but is not allowed to directly refer to the private members of `ContactDir`. The idea is that private members should be completely hidden so that they cannot be tampered with by the class clients.

This restriction may prove too prohibitive for classes from which other classes are likely to be derived. Denying the derived class access to the base class private members may convolute its implementation or even make it impractical to define.

The restriction can be relaxed by defining the base class private members as protected instead. As far as the clients of a class are concerned, a protected member is the same as a private member: it cannot be accessed by the class clients. However, a protected base class member can be accessed by any class derived from it.

For example, the private members of `ContactDir` can be made protected by substituting the keyword `protected` for `private`:

```
class ContactDir {
        //...
protected:
        int     Lookup      (const char *name);
        Contact **contacts; // list of contacts
        int     dirSize;    // current directory size
        int     maxSize;    // max directory size
};
```

As a result, `Lookup` and the data members of `ContactDir` are now accessible to `SmartDir`.

The access keywords `private`, `public`, and `protected` can occur as many times as desired in a class definition. Each access keyword specifies the access characteristics of the members following it until the next access keyword:

```
class Foo {
    public:
        // public members...
    private:
        // private members...
    protected:
        // protected members...
    public:
        // more public members...
    protected:
        // more protected members...
};
```

□

Private, Public, and Protected Base Classes

A base class may be specified to be private, public, or protected. Unless so specified, the base class is assumed to be private:

```
class A {
    private:    int x;      void Fx (void);
    public:     int y;      void Fy (void);
    protected:  int z;      void Fz (void);
};
class B : A {};             // A is a private base class of B
class C : private A {};     // A is a private base class of C
class D : public A {};      // A is a public base class of D
class E : protected A {};   // A is a protected base class of E
```

The behavior of these is as follows (see Table 8.1 for a summary):

- All the members of a private base class become *private* members of the derived class. So `x`, `Fx`, `y`, `Fy`, `z`, and `Fz` all become private members of `B` and `C`.
- The members of a public base class keep their access characteristics in the derived class. So, `x` and `Fx` becomes private members of `D`, `y` and `Fy` become public members of `D`, and `z` and `Fz` become protected members of `D`.
- The private members of a protected base class become *private* members of the derived class. Whereas, the public and protected members of a protected base class become *protected* members of the derived class. So, `x` and `Fx` become private members of `E`, and `y`, `Fy`, `z`, and `Fz` become protected members of `E`.

Table 8.1 Base class access inheritance rules.

Base Class	Private Derived	Public Derived	Protected Derived
Private Member	*private*	*private*	*private*
Public Member	*private*	*public*	*protected*
Protected Member	*private*	*protected*	*protected*

It is also possible to individually exempt a base class member from the access changes specified by a derived class, so that it retains its original access characteristics. To do this, the exempted member is fully named in the derived class under its original access characteristic. For example:

```
class C : private A {
    //...
public:     A::Fy;      // makes Fy a public member of C
protected:  A::z;       // makes z a protected member of C
};
```

□

Consider another variation of the `ContactDir` class, called `SortedDir`, which ensures that new contacts are inserted in such a manner that the list remains sorted at all times. The obvious advantage of this is that the search speed can be improved by using the binary search algorithm instead of linear search.

The actual search is performed by the `Lookup` member function. Therefore we need to redefine this function in `SortedDir` so that it uses the binary search algorithm. However, all the other member functions refer to `ContactDir::Lookup`. We can also redefine these so that they refer to `SortedDir::Lookup` instead. If we follow this approach, the value of inheritance becomes rather questionable, because we would have practically redefined the whole class.

What we really want to do is to find a way of expressing this: `Lookup` should be tied to the *type* of the object which invokes it. If the object is of type `SortedDir` then invoking `Lookup` (from anywhere, even from within the member functions of `ContactDir`) should mean `SortedDir::Lookup`. Similarly, if the object is of type `ContactDir` then calling `Lookup` (from anywhere) should mean `ContactDir::Lookup`.

This can be achieved through the **dynamic binding** of `Lookup`: the decision as to which version of `Lookup` to call is made at runtime depending on the type of the object.

In C++, dynamic binding is supported by virtual member functions. A member function is declared as virtual by inserting the keyword `virtual` before its prototype in the base class. Any member function, including constructors and destructors, can be declared as virtual. `Lookup` should be declared as virtual in `ContactDir`:

```
class ContactDir {
    //...
protected:
    virtual int Lookup (const char *name);
    //...
};
```

Only nonstatic member functions can be declared as virtual. A virtual function redefined in a derived class must have exactly the same prototype as the one in the base class. Virtual functions can be overloaded like other member functions.

Listing 8.3 shows the definition of `SortedDir` as a derived class of `ContactDir`.

Listing 8.3


```
1  class SortedDir : public ContactDir {
2  public:
3                  SortedDir   (const int max) : ContactDir(max) {}
4  protected:
5      virtual int Lookup      (const char *name);
6  };
```

Annotation

3 The constructor simply invokes the base class constructor.

5 `Lookup` is again declared as virtual to enable any class derived from `SortedDir` to redefine it.

The new definition of `Lookup` is as follows:

```
int SortedDir::Lookup (const char *name)
{
    int bot = 0;
    int top = dirSize - 1;
    int pos = 0;
    int    mid, cmp;

    while (bot <= top) {
       mid = (bot + top) / 2;
       if ((cmp = strcmp(name, contacts[mid]->Name())) == 0)
           return mid;             // return item index
       else if (cmp < 0)
           pos = top = mid - 1;    // restrict search to lower half
       else
           pos = bot = mid + 1;    // restrict search to upper half
    }
    return pos < 0 ? 0 : pos;      // expected slot
}
```

The following code fragment illustrates that `SortedDir::Lookup` is called by `ContactDir::Insert` when invoked via a `SortedDir` object:

```
SortedDir   dir(10);
dir.Insert(Contact("Mary", "11 South Rd", "282 1324"));
dir.Insert(Contact("Peter", "9 Port Rd", "678 9862"));
dir.Insert(Contact("Jane", "321 Yara Ln", "982 6252"));
dir.Insert(Contact("Jack", "42 Wayne St", "663 2989"));
dir.Insert(Contact("Fred", "2 High St", "458 2324"));
cout << dir;
```

It will produce the following output:

```
(Fred , 2 High St , 458 2324)
(Jack , 42 Wayne St , 663 2989)
(Jane , 321 Yara Ln , 982 6252)
(Mary , 11 South Rd , 282 1324)
(Peter , 9 Port Rd , 678 9862)
```

□

Multiple Inheritance

The derived classes encountered so far in this chapter represent **single inheritance**, because each inherits its attributes from a single base class. Alternatively, a derived class may have multiple base classes. This is referred to as **multiple inheritance**.

For example, suppose we have defined two classes for, respectively, representing lists of options and bitmapped windows:

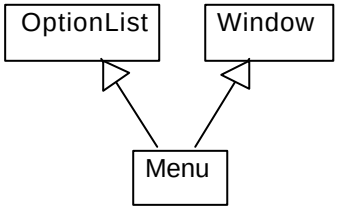
```
class OptionList {
public:
    OptionList  (int n);
    ~OptionList (void);
    //...
};

class Window {
public:
    Window  (Rect &bounds);
    ~Window (void);
    //...
};
```

A menu is a list of options displayed within its own window. It therefore makes sense to define `Menu` by deriving it from `OptionList` and `Window`:

```
class Menu : public OptionList, public Window {
public:
    Menu    (int n, Rect &bounds);
    ~Menu   (void);
    //...
};
```

Under multiple inheritance, a derived class inherits all of the members of its base classes. As before, each of the base classes may be private, public, or protected. The same base member access principles apply. Figure 8.4 illustrates the class hierarchy for `Menu`.

Figure 8.4 The Menu class hierarchy



Since the base classes of `Menu` have constructors that take arguments, the constructor for the derived class should invoke these in its member initialization list:

```
Menu::Menu (int n, Rect &bounds) : OptionList(n), Window(bounds)
{
    //...
}
```

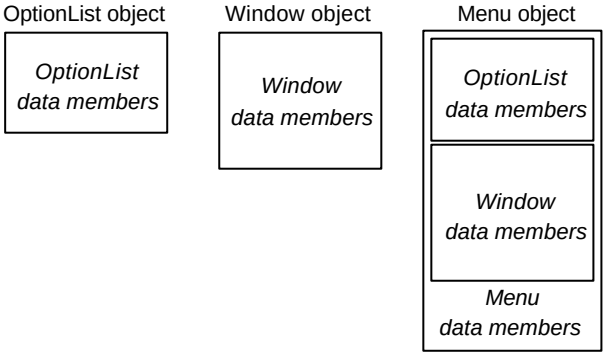
The order in which the base class constructors are invoked is the same as the order in which they are specified in the derived class header (*not* the order in which they appear in the derived class constructor's member initialization list). For `Menu`, for example, the constructor for `OptionList` is invoked before the constructor for `Window`, even if we change their order in the constructor:

```
Menu::Menu (int n, Rect &bounds) : Window(bounds), OptionList(n)
{
    //...
}
```

The destructors are applied in the reverse order: `~Menu`, followed by `~Window`, followed by `~OptionList`.

The obvious implementation of a derived class object is to contain one object from each of its base classes. Figure 8.5 illustrates the relationship between a `Menu` object and its base class objects.

Figure 8.5 Base and derived class objects.



In general, a derived class may have any number of base classes, all of which must be distinct:

```
class X : A, B, A {     // illegal: A appears twice
    //...
};
```

□

Ambiguity

Multiple inheritance further complicates the rules for referring to the members of a class. For example, suppose that both `OptionList` and `Window` have a member function called `Highlight` for highlighting a specific part of either object type:

```
class OptionList {
public:
    //...
    void Highlight (int part);
};

class Window {
public:
    //...
    void Highlight (int part);
};
```

The derived class `Menu` will inherit both these functions. As a result, the call

```
m.Highlight(0);
```

(where `m` is a `Menu` object) is ambiguous and will not compile, because it is not clear whether it refers to `OptionList::Highlight` or `Window::Highlight`. The ambiguity is resolved by making the call explicit:

```
m.Window::Highlight(0);
```

Alternatively, we can define a `Highlight` member for `Menu` which in turn calls the `Highlight` members of the base classes:

```
class Menu : public OptionList, public Window {
public:
    //...
    void Highlight (int part);
};

void Menu::Highlight (int part)
{
    OptionList::Highlight(part);
    Window::Highlight(part);
}
```

□

Type Conversion

For any derived class there is an implicit type conversion from the derived class to any of its *public* base classes. This can be used for converting a derived class object to a base class object, be it a proper object, a reference, or a pointer:

```
Menu    menu(n, bounds);
Window  win = menu;
Window  &wRef = menu;
Window  *wPtr = &menu;
```

Such conversions are safe because the derived class object always contains all of its base class objects. The first assignment, for example, causes the `Window` component of `menu` to be assigned to `win`.

By contrast, there is no implicit conversion from a base class to a derived class. The reason being that such a conversion is potentially dangerous due to the fact that the derived class object may have data members not present in the base class object. The extra data members will therefore end up with unpredictable values. All such conversions must be explicitly cast to confirm the programmer's intention:

```
Menu    &mRef = (Menu&) win;        // caution!
Menu    *mPtr = (Menu*) &win;       // caution!
```

A base class object cannot be assigned to a derived class object unless there is a type conversion constructor in the derived class defined for this purpose. For example, given

```
class Menu : public OptionList, public Window {
public:
    //...
    Menu (Window&);
};
```

the following would be valid and would use the constructor to convert `win` to a `Menu` object before assigning:

```
menu = win;     // invokes Menu::Menu(Window&)
```

□

Consider the problem of recording the average time required for a message to be transmitted from one machine to another in a long-haul network. This can be represented as a table, as illustrated by Table 8.2.

Table 8.2 Message transmission time (in seconds).

	Sydney	Melbourne	Perth
Sydney	0.00	3.55	12.45
Melbourne	2.34	0.00	10.31
Perth	15.36	9.32	0.00

The row and column indices for this table are strings rather than integers, so the `Matrix` class (Chapter 7) will not be adequate for representing the table. We need a way of mapping strings to indices. This is already supported by the `AssocVec` class (Chapter 7). As shown in Listing 8.4, `Table1` can be defined as a derived class of `Matrix` and `AssocVec`.

Listing 8.4

```
1  class Table1 : Matrix, AssocVec  {
2  public:
3              Table1      (const short entries)
4                              : Matrix(entries, entries),
5                                AssocVec(entries)     {}
6      double& operator () (const char *src, const char *dest);
7  };

8  double& Table1::operator () (const char *src, const char *dest)
9  {
10     return  this->Matrix::operator()(
11                     this->AssocVec::operator[](src),
12                     this->AssocVec::operator[](dest)
13             );
14 }
```

Here is a simple test of the class

```
Table tab(3);
tab("Sydney","Perth") = 12.45;
cout << "Sydney -> Perth = " << tab("Sydney","Perth") << '\n';
```

which produces the following output:

```
Sydney -> Perth = 12.45
```

Another way of defining this class is to derive it from `Matrix` and include an `AssocVec` object as a data member (see Listing 8.5).

Listing 8.5


```
 1  class Table2 : Matrix  {
 2  public:
 3              Table2      (const short entries)
 4                              : Matrix(entries, entries),
 5                                index(entries)        {}
 6      double& operator () (const char *src, const char *dest);
 7  private:
 8      AssocVec index;     // row and column index
 9  };

10  double& Table2::operator () (const char *src, const char *dest)
11  {
12      return  this->Matrix::operator()(index[src], index[dest]);
13  }
```

The inevitable question is: which one is a better solution, `Table1` or `Table2`? The answer lies in the relationship of table to matrix and associative vector:

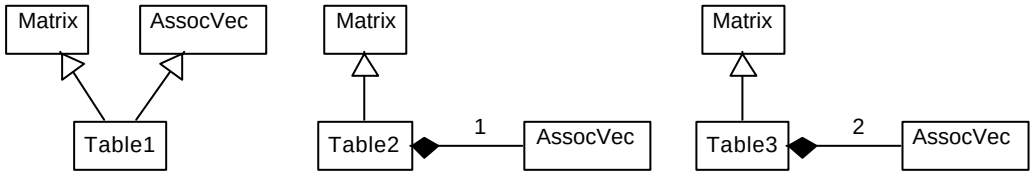
- A table *is a* form of matrix.
- A table is not an associative vector, but rather *uses an* associative vector to manage the association of its row and column labels with positional indexes.

In general, an *is-a* relationship is best realized using inheritance, because it implies that the properties of one object are shared by another object. On the other hand, a *uses-a* (or *has-a*) relationship is best realized using composition, because it implies that one object is contained by another object. `Table2` is therefore the preferred solution.

It is worth considering which of the two versions of table better lends itself to generalization. One obvious generalization is to remove the restriction that the table should be square, and to allow the rows and columns to have different labels. To do this, we need to provide two sets of indexes: one for rows and one for columns. Hence we need two associative vectors. It is arguably easier to expand `Table2` to do this rather than modify `Table1` (see Listing 8.6).

Figure 8.6 shows the class hierarchies for the three variations of table.

Figure 8.6 Variations of table.



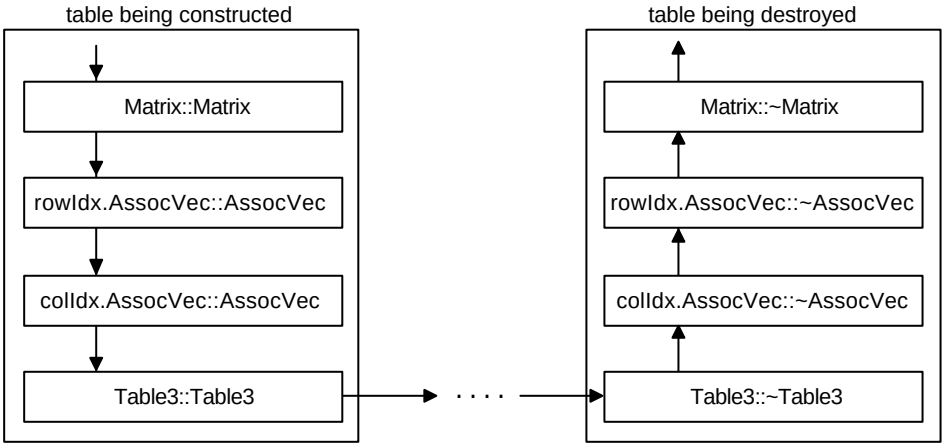
Listing 8.6

```
 1  class Table3 : Matrix {
 2  public:
 3              Table3      (const short rows, const short cols)
 4                              : Matrix(rows,cols),
 5                                rowIdx(rows),
 6                                colIdx(cols)      {}
 7      double& operator () (const char *src, const char *dest);

 8  private:
 9      AssocVec rowIdx;    // row index
10      AssocVec colIdx;    // column index
11  };

12  double& Table3::operator () (const char *src, const char *dest)
13  {
14      return this->Matrix::operator()(rowIdx[src], colIdx[dest]);
15  }
```

For a derived class which also has class object data members, the order of object construction is as follows. First the base class constructors are invoked in the order in which they appear in the derived class header. Then the class object data members are initialized by their constructors being invoked in the same order in which they are declared in the class. Finally, the derived class constructor is invoked. As before, the derived class object is destroyed in the reverse order of construction.

Figure 8.7 illustrates this for a `Table3` object.

Figure 8.7 Table3 object construction and destruction order.



□

Virtual Base Classes

Recall the `Menu` class and suppose that its two base classes are also multiply derived:

```
class OptionList : public Widget, List              { /*...*/ };
class Window     : public Widget, Port              { /*...*/ };
class Menu       : public OptionList, public Window  { /*...*/ };
```

Since `Widget` is a base class for both `OptionList` and `Window`, each menu object will have two widget objects (see Figure 8.8a). This is not desirable (because a menu is considered a single widget) and may lead to ambiguity. For example, when applying a widget member function to a menu object, it is not clear as to which of the two widget objects it should be applied. The problem is overcome by making `Widget` a **virtual** base class of `OptionList` and `Window`. A base class is made virtual by placing the keyword `virtual` before its name in the derived class header:

```
class OptionList : virtual public Widget, List      { /*...*/ };
class Window     : virtual public Widget, Port      { /*...*/ };
```

This ensures that a `Menu` object will contain exactly one `Widget` object. In other words, `OptionList` and `Window` will share the same `Widget` object.

An object of a class which is derived from a virtual base class does not directly contain the latter's object, but rather a pointer to it (see Figure 8.8b and 8.8c). This enables multiple occurrences of a virtual class in a hierarchy to be collapsed into one (see Figure 8.8d).

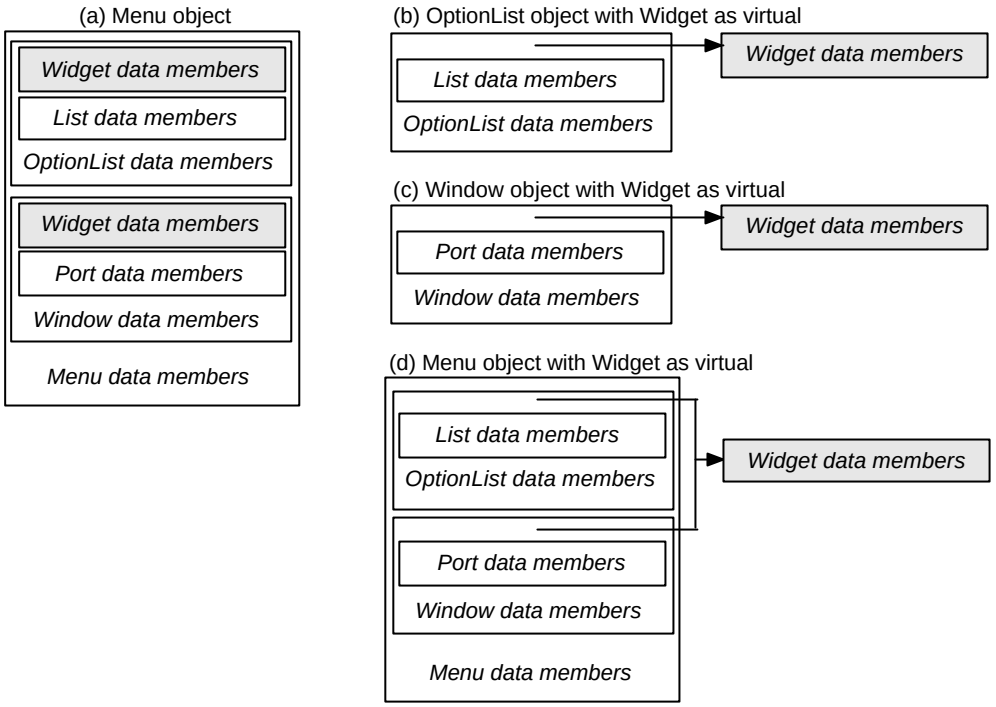
If in a class hierarchy some instances of a base class X are declared as virtual and other instances as nonvirtual, then the derived class object will contain an X object for each nonvirtual instance of X, and a single X object for all virtual occurrences of X.

A virtual base class object is initialized, not necessarily by its immediate derived class, but by the derived class farthest down the class hierarchy. This rule ensures that the virtual base class object is initialized only once. For example, in a menu object, the widget object is initialized by the `Menu` constructor (which overrides the invocation of the `Widget` constructor by `OptionList` or `Window`):

```
Menu::Menu (int n, Rect &bounds) :  Widget(bounds),
                                    OptionList(n),
                                    Window(bounds)
{ //... }
```

Regardless of where it appears in a class hierarchy, a virtual base class object is always constructed *before* nonvirtual objects in the same hierarchy.

Figure 8.8 Nonvirtual and virtual base classes.



If in a class hierarchy a virtual base is declared with conflicting access characteristics (i.e., any combination of private, protected, and public), then the most accessible will dominate. For example, if `Widget` were declared a private base class of `OptionList`, and a public base class of `Window`, then it would still be a public base class of `Menu`.

□

Overloaded Operators

Except for the assignment operator, a derived class inherits all the overloaded operators of its base classes. An operator overloaded by the derived class itself hides the overloading of the same operator by the base classes (in exactly the same way member functions of a derived class hide member functions of base classes).

Memberwise initialization and assignment (see Chapter 7) extend to derived classes. For any given class `Y` derived from `X`, memberwise initialization is handled by an automatically-generated (or user-defined) constructor of the form:

```
Y::Y (const Y&);
```

Similarly, memberwise assignment is handled by an automatically-generated (or user-defined) overloading of the `=` operator:

```
Y& Y::operator = (Y&)
```

Memberwise initialization (or assignment) of a derived class object involves the memberwise initialization (or assignment) of its base classes as well as its class object members.

Special care is needed when a derived class relies on the overloading of `new` and `delete` operators for its base class. For example, recall the overloading of these two operators for the `Point` class in Chapter 7, and suppose that we wish to use them for a derived class:

```
class Point3D : public Point {
public:
    //...
private:
    int depth;
};
```

Because the implementation of `Point::operator new` assumes that the requested block should be the size of a `Point` object, its inheritance by the `Point3D` class leads to a problem: it fails to account for the extra space needed by the data member of the latter (i.e., `depth`).

To avoid this problem, an overloading of `new` should attempt to allocate the exact amount of storage specified by its size parameter, rather than assuming a predefined size. Similarly, an overloading of `delete` should note the size specified by its second parameter and attempt to release exactly those many bytes.

□

Exercises

8.1 Consider a `Year` class which divides the days in a year into work days and off days. Because each day has a binary value, `Year` is easily derived from `BitVec`:

```
enum Month {
    Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec
};

class Year : public BitVec {
public:
            Year    (const short year);
    void    WorkDay (const short day);  // set day as work day
    void    OffDay  (const short day);  // set day as off day
    Bool    Working (const short day);  // true if a work day
    short   Day     (const short day,   // convert date to day
                     const Month month, const short year);
protected:
    short   year;   // calendar year
};
```

Days are sequentially numbered from the beginning of the year, starting at 1 for January 1st. Complete the `Year` class by implementing its member functions.

8.2 Consider an educational application program which given an arbitrary set of values, $X = [x_1, x_2, ...,x_n]$, generates a set of n linear equations whose solution is X, and then proceeds to illustrate this by solving the equations using Gaussian elimination. Derive a class named `LinearEqns` from `Matrix` and for this purpose and define the following member functions for it:

- A constructor which accepts X as a matrix, and a destructor.
- `Generate` which randomly generates a system of linear equations as a matrix M. It should take a positive integer (`coef`) as argument and generate a set of equations, ensuring that the range of the coefficients does not exceed `coef`. Use a random number generator (e.g., `random` under UNIX) to generate the coefficients. To ensure that X is a solution for the equations denoted by M, the last element of a row k is denoted by:

$$M[k,n+1] = \sum_{i=1}^{n} M[k,i] \times X[i]$$

- `Solve` which uses Gaussian elimination to solve the equations generated by `Generate`. `Solve` should the output operator of `Matrix` to display the augmented matrix each time the elements below a pivot are eliminated.

8.3 Enumerations introduced by an enum declaration are small subsets of integers. In certain applications we may need to construct sets of such enumerations. For example, in a parser, each parsing routine may be passed a set of symbols that

should not be skipped when the parser attempts to recover from a syntax error. These symbols are typically the reserved words of the language:

```
enum Reserved {classSym, privateSym, publicSym, protectedSym,
               friendSym, ifSym, elseSym, switchSym,...};
```

Given that there may be at most *n* elements in a set (*n* being a small number) the set can be efficiently represented as a bit vector of *n* elements. Derive a class named `EnumSet` from `BitVec` to facilitate this. `EnumSet` should overload the following operators:

- Operator `+` for set union.
- Operator `-` for set difference.
- Operator `*` for set intersection.
- Operator `%` for set membership.
- Operators `<=` and `>=` for testing if a set is a subset of another.
- Operators `>>` and `<<` for, respectively, adding an element to and removing an element from a set.

8.4 An **abstract class** is a class which is never used directly but provides a skeleton for other classes to be derived from it. Typically, all the member functions of an abstract are virtual and have dummy implementations. The following is a simple example of an abstract class:

```
class Database {
public:
    virtual void    Insert  (Key, Data)     {}
    virtual void    Delete  (Key)           {}
    virtual Data    Search  (Key)           {return 0;}
};
```

It provides a skeleton for a database-like classes. Examples of the kind of classes which could be derived from database include: linked-list, binary tree, and B-tree. First derive a B-tree class from Database and then derive a B*-tree from B-tree:

```
class BTree : public Database { /*...*/ };
class BStar : public BTree    { /*...*/ };
```

See Comer (1979) for a description of B-tree and B*-tree. For the purpose of this exercise, use the built-in type `int` for `Key` and `double` for `Data`.

□

9. Templates

This chapter describes the template facility of C++ for defining functions and classes. Templates facilitate the *generic* definition of functions and classes so that they are not tied to specific implementation types. They are invaluable in that they dispense with the burden of redefining a function or class so that it will work with yet another data type.

A function template defines an **algorithm**. An algorithm is a generic recipe for accomplishing a task, independent of the particular data types used for its implementation. For example, the binary search algorithm operates on a sorted array of items, whose exact type is irrelevant to the algorithm. Binary search can therefore be defined as a function template with a type parameter which denotes the type of the array items. This template then becomes a blueprint for generating executable functions by substituting a concrete type for the type parameter. This process is called **instantiation** and its outcome is a conventional function.

A class template defines a **parameterized type**. A parameterized type is a data type defined in terms of other data types, one or more of which are unspecified. Most data types can be defined independently of the concrete data types used in their implementation. For example, the stack data type involves a set of items whose exact type is irrelevant to the concept of stack. Stack can therefore be defined as a class template with a type parameter which specifies the type of the items to be stored on the stack. This template can then be instantiated, by substituting a concrete type for the type parameter, to generate executable stack classes.

Templates provide direct support for writing reusable code. This in turn makes them an ideal tool for defining generic libraries.

We will present a few simple examples to illustrate how templates are defined, instantiated, and specialized. We will describe the use of nontype parameters in class templates, and discuss the role of class members, friends, and derivations in the context of class templates.

Function Template Definition

A function template definition (or declaration) is always preceded by a **template clause**, which consists of the keyword `template` and a list of one or more type parameters. For example,

```
template <class T>  T Max (T, T);
```

declares a function template named `Max` for returning the maximum of two objects. `T` denotes an unspecified (generic) type. `Max` is specified to compare two objects of the same type and return the larger of the two. Both arguments and the return value are therefore of the same type `T`. The definition of a function template is very similar to a normal function, except that the specified type parameters can be referred to within the definition. The definition of `Max` is shown in Listing 9.1.

Listing 9.1

```
1  template <class T>
2  T Max (T val1, T val2)
3  {
4      return val1 > val2 ? val1 : val2;
5  }
```

A type parameter is an arbitrary identifier whose scope is limited to the function itself. Type parameters always appear inside `<>`. Each type parameter consists of the keyword `class` followed by the parameter name. When multiple type parameters are used, they should be separated by commas. Each specified type parameter must actually be referred to in the function prototype. The keyword `class` cannot be factored out:

```
template <class T1, class T2, class T3>
    T3 Relation(T1, T2);        // ok

template <class T1, class T2>
    int Compare (T1, T1);       // illegal! T2 not used.

template <class T1, T2>         // illegal! class missing for T2
    int Compare (T1, T2);
```

For static, inline, and extern functions, the respective keyword must appear after the template clause, and not before it:

```
template <class T>
    inline T Max (T val1, T val2);  // ok

inline template <class T>           // illegal! inline misplaced
    T Max (T val1, T val2);
```

□

Function Template Instantiation

A function template represents an algorithm from which executable implementations of the function can be generated by binding its type parameters to concrete (built-in or user-defined) types. For example, given the earlier template definition of `Max`, the code fragment

```
cout << Max(19, 5) << ' '
     << Max(10.5, 20.3) << ' '
     << Max('a','b') << '\n';
```

will produce the following output:

```
19 20.3 b
```

In the first call to `Max`, both arguments are integers, hence `T` is bound to `int`. In the second call, both arguments are reals, hence `T` is bound to `double`. In the final call, both arguments are characters, hence `T` is bound to `char`. A total of three functions are therefore generated by the compiler to handle these cases:

```
int     Max (int, int);
double  Max (double, double);
char    Max (char, char);
```

When the compiler encounters a call to a template function, it attempts to infer the concrete type to be substituted for each type parameter by examining the type of the arguments in the call. The compiler does *not* attempt any implicit type conversions to ensure a match. As a result, it cannot resolve the binding of the same type parameter to reasonable but unidentical types. For example:

```
Max(10, 12.6);
```

would be considered an error because it requires the first argument to be converted to `double` so that both arguments can match `T`. The same restriction even applies to the ordinary parameters of a function template. For example, consider the alternative definition of `Max` in Listing 9.2 for finding the maximum value in an array of values. The ordinary parameter `n` denotes the number of array elements. A matching argument for this parameter must be of type `int`:

```
unsigned nValues = 4;
double   values[] = {10.3, 19.5, 20.6, 3.5};
Max(values, 4);          // ok
Max(values, nValues);    // illegal! nValues does not match int
```

Listing 9.2


```
1  template <class T>
2  T Max (T *vals, int n)
3  {
4      T max = vals[0];
5      for (register i = 1; i < n; ++i)
6          if (vals[i] > max)
7              max = vals[i];
8      return max;
9  }
```

The obvious solution to both problems is to use explicit type conversion:

```
Max(double(10), 12.6);
Max(values, int(nValues));
```

As illustrated by Listings 9.1 and 9.2, function templates can be **overloaded** in exactly the same way as normal functions. The same rule applies: each overloaded definition must have a unique signature.

Both definitions of `Max` assume that the `>` operator is defined for the type substituted in an instantiation. When this is not the case, the compiler flags it as an error:

```
Point pt1(10,20), pt2(20,30);
Max(pt1, pt2);          // illegal: pt1 > pt2 undefined
```

For some other types, the operator may be defined but not produce the desired effect. For example, using `Max` to compare two strings will result in their pointer values being compared, not their character sequences:

```
Max("Day", "Night");    // caution: "Day" > "Night" undesirable
```

This case can be correctly handled through a **specialization** of the function, which involves defining an instance of the function to exactly match the proposed argument types:

```
#include <string.h>
char* Max (char *str1, char *str2)      // specialization of Max
{
    return strcmp(str1, str2) > 0 ? str1 : str2;
}
```

Given this specialization, the above call now matches this function and will *not* result in an instance of the function template to be instantiated for `char*`.

□

Example: Binary Search

Recall the binary search algorithm implemented in Chapter 5. Binary search is better defined as a function template so that it can be used for searching arrays of any type. Listing 9.3 provides a template definition.

Listing 9.3

```
 1  template <class Type>
 2  int BinSearch (Type &item, Type *table, int n)
 3  {
 4      int bot = 0;
 5      int top = n - 1;
 6      int mid, cmp;

 7      while (bot <= top) {
 8          mid = (bot + top) / 2;
 9          if (item == table[mid])
10              return mid;             // return item index
11          else if (item < table[mid])
12              top = mid - 1;          // restrict search to lower half
13          else
14              bot = mid + 1;          // restrict search to upper half
15      }
16      return -1;                      // not found
17  }
```

Annotation

3 This is the template clause. It introduces `Type` as a type parameter, the scope for which is the entire definition of the `BinSearch` function.

4 `BinSearch` searches for an item denoted by `item` in the sorted array denoted by `table`, the dimension for which is denoted by `n`.

9 This line assumes that the operator `==` is defined for the type to which `Type` is bound in an instantiation.

11 This line assumes that the operator `<` is defined for the type to which `Type` is bound in an instantiation.

Instantiating `BinSearch` with `Type` bound to a built-in type such as `int` has the desired effect. For example,

```
int nums[] = {10, 12, 30, 38, 52, 100};
cout << BinSearch(52, nums, 6) << '\n';
```

produces the expected output:

```
4
```


Now let us instantiate `BinSearch` for a user-defined type such as `RawBook` (see Chapter 7). First, we need to ensure that the comparison operators are defined for our user-defined type:

```
class RawBook {
public:
    //...
    int     operator <  (RawBook &b)    {return Compare(b) < 0;}
    int     operator >  (RawBook &b)    {return Compare(b) > 0;}
    int     operator == (RawBook &b)    {return Compare(b) == 0;}
private:
    int     Compare     (RawBook&);
    //...
};

int RawBook::Compare (RawBook &b)
{
    int cmp;
    Book *b1 = RawToBook();
    Book *b2 = b.RawToBook();
    if ((cmp = strcmp(b1->title, b2->title)) == 0)
        if ((cmp = strcmp(b1->author, b2->author)) == 0)
            return strcmp(b1->publisher, b2->publisher);
    return cmp;
}
```

All are defined in terms of the private member function `Compare` which compares two books by giving priority to their titles, then authors, and finally publishers. The code fragment

```
RawBook books[] = {
   RawBook("%APeters\0%TBlue
Earth\0%PPhedra\0%CSydney\0%Y1981\0\n"),
   RawBook("%TPregnancy\0%AJackson\0%Y1987\0%PMiles\0\n"),
   RawBook("%TZoro\0%ASmiths\0%Y1988\0%PMiles\0\n")
};
cout << BinSearch(RawBook("%TPregnancy\0%AJackson\0%PMiles\0\n"),
                  books, 3) << '\n';
```

produces the output

```
1
```

which confirms that `BinSearch` is instantiated as expected.

□

Class Template Definition

A class template definition (or declaration) is always preceded by a template clause. For example,

```
template <class Type> class Stack;
```

declares a class template named `Stack`. A class template clause follows the same syntax rules as a function template clause.

The definition of a class template is very similar to a normal class, except that the specified type parameters can be referred to within the definition. The definition of `Stack` is shown in Listing 9.4.

Listing 9.4

```
 1  template <class Type>
 2  class Stack {
 3  public:
 4              Stack   (int max) : stack(new Type[max]),
 5                                  top(-1), maxSize(max) {}
 6              ~Stack  (void)      {delete [] stack;}
 7      void    Push    (Type &val);
 8      void    Pop     (void)      {if (top >= 0) --top;}
 9      Type&   Top     (void)      {return stack[top];}
10      friend  ostream& operator << (ostream&, Stack&);
11  private:
12      Type        *stack;     // stack array
13      int         top;        // index of top stack entry
14      const int   maxSize;    // max size of stack
15  };
```

The member functions of `Stack` are defined inline except for `Push`. The `<<` operator is also overloaded to display the stack contents for testing purposes. These two are defined as follows:

```
template <class Type>
void Stack<Type>::Push (Type &val)
{
    if (top+1 < maxSize)
        stack[++top] = val;
}

template <class Type>
ostream& operator << (ostream& os, Stack<Type>& s)
{
    for (register i = 0; i <= s.top; ++i)
        os << s.stack[i] << " ";
    return os;
}
```

Except for within the class definition itself, a reference to a class template must include its template parameter list. This is why the definition of `Push` and << use the name `Stack<Type>` instead of `Stack`. □

Class Template Instantiation

A class template represents a generic class from which executable implementations of the class can be generated by binding its type parameters to concrete (built-in or user-defined) types. For example, given the earlier template definition of `Stack`, it is easy to generate stacks of a variety of types through instantiation:

```
Stack<int>    s1(10);       // stack of integers
Stack<double> s2(10);       // stack of doubles
Stack<Point>  s3(10);       // stack of points
```

Each of these instantiations causes the member functions of the class to be accordingly instantiated. So, for example, in the first instantiation, the member functions will be instantiated with `Type` bounds to `int`. Therefore,

```
s1.Push(10);
s1.Push(20);
s1.Push(30);
cout << s1 << '\n';
```

will produce the following output:

```
10 20 30
```

When a nontemplate class or function refers to a class template, it should bind the latter's type parameters to *defined* types. For example:

```
class Sample {
    Stack<int>  intStack;   // ok
    Stack<Type> typeStack;  // illegal! Type is undefined
    //...
};
```

The combination of a class template and arguments for all of its type parameters (e.g., `Stack<int>`) represents a valid type specifier. It may appear wherever a C++ type may appear.

If a class template is used as a part of the definition of another class template (or function template), then the former's type parameters can be bound to the latter's template parameters. For example:

```
template <class Type>
class Sample {
    Stack<int>  intStack;   // ok
    Stack<Type> typeStack;  // ok
    //...
};
```

□

Nontype Parameters

Unlike a function template, not all parameters of a class template are required to represents types. Value parameters (of defined types) may also be used. Listing 9.5 shows a variation of the `Stack` class, where the maximum size of the stack is denoted by a template parameter (rather than a data member).

Listing 9.5

```
 1  template <class Type, int maxSize>
 2  class Stack {
 3  public:
 4              Stack   (void) :    stack(new Type[maxSize]), top(-1) {}
 5              ~Stack  (void)      {delete [] stack;}
 6      void    Push    (Type &val);
 7      void    Pop     (void)      {if (top >= 0) --top;}
 8      Type    &Top    (void)      {return stack[top];}
 9  private:
10      Type    *stack;     // stack array
11      int     top;        // index of top stack entry
12  };
```

Both parameters are now required for referring to `Stack` outside the class. For example, `Push` is now defined as follows:

```
template <class Type, int maxSize>
void Stack<Type, maxSize>::Push (Type &val)
{
    if (top+1 < maxSize)
        stack[++top] = val;
}
```

Unfortunately, the operator `<<` cannot be defined as before, since value template parameters are not allowed for nonmember functions:

```
template <class Type, int maxSize>      // illegal!
ostream &operator << (ostream&, Stack<Type, maxSize>&);
```

Instantiating the `Stack` template now requires providing two arguments: a defined type for `Type` and a defined integer value for `maxSize`. The type of the value must match the type of value parameter exactly. The value itself must be a constant expression which can be evaluated at compile-time. For example:

```
Stack<int, 10>  s1;     // ok
Stack<int, 10u> s2;     // illegal! 10u doesn't match int
Stack<int, 5+5> s3;     // ok
int n = 10;
Stack<int, n>   s4;     // illegal! n is a run-time value
```

□

Class Template Specialization

The algorithms defined by the member functions of a class template may be inappropriate for certain types. For example, instantiating the `Stack` class with the type `char*` may lead to problems because the `Push` function will simply push a string pointer onto the stack without copying it. As a result, if the original string is destroyed the stack entry will be invalid.

Such cases can be properly handled by specializing the inappropriate member functions. Like a global function template, a member function of a class template is specialized by providing an implementation of it based on a particular type. For example,

```
void Stack<char*>::Push (char* &val)
{
    if (top+1 < maxSize) {
        stack[++top] = new char[strlen(val) + 1];
        strcpy(stack[top], val);
    }
}
```

specializes the `Push` member for the `char*` type. To free the allocated storage, `Pop` needs to be specialized as well:

```
void Stack<char*>::Pop (void)
{
    if (top >= 0)
        delete stack[top--];
}
```

It is also possible to specialize a class template as a whole, in which case all the class members must be specialized as a part of the process:

```
typedef char* Str;
class Stack<Str> {
public:
            Stack<Str>::Stack   (int max) : stack(new Str[max]),
                                top(-1), maxSize(max) {}
            ~Stack  (void)      {delete [] stack;}
    void    Push    (Str val);
    void    Pop     (void);
    Str     Top     (void)      {return stack[top];}
    friend  ostream& operator << (ostream&, Stack<Str>&);
private:
    Str         *stack;     // stack array
    int         top;        // index of top stack entry
    const int   maxSize;    // max size of stack
};
```

Although the friend declaration of `<<` is necessary, because this is a nonmember function, its earlier definition suffices. □

Class Template Members

A class template may have constant, reference, and static members just like an ordinary class. The use of constant and reference members is exactly as before. Static data members are shared by the objects of an instantiation. There will therefore be an instance of each static data member per instantiation of the class template.

As an example, consider adding a static data member to the `Stack` class to enable `Top` to return a value when the stack is empty:

```
template <class Type>
class Stack {
public:
    //...
    Type&   Top (void);
private:
    //...
    static  Type    dummy;      // dummy entry
};

template <class Type>
Type& Stack<Type>::Top (void)
{
    return top >= 0 ? stack[top] : dummy;

}
```

There are two ways in which a static data member can be initialized: as a template or as a specific type. For example,

```
template <class Type> Type Stack<Type>::dummy = 0;
```

provides a template initialization for `dummy`. This is instantiated for each instantiation of `Stack`. (Note, however, that the value 0 may be inappropriate for non-numeric types).

Alternatively, an explicit instance of this initialization may be provided for each instantiation of `Stack`. A `Stack<int>` instantiation, for example, could use the following initialization of `dummy`:

```
int Stack<int>::dummy = 0;
```

□

Class Template Friends

When a function or class is declared as a friend of a class template, the friendship can take one of there forms, as illustrated by the following example.

Consider the `Stack` class template and a function template named `Foo`:

```
template <class T> void Foo (T&);
```

We wish to define a class named `Sample` and declare `Foo` and `Stack` as its friends. The following makes a *specific* instance of `Foo` and `Stack` friends of *all* instances of `Sample`:

```
template <class T>
class Sample {              // one-to-many friendship
    friend Foo<int>;
    friend Stack<int>;
    //...
};
```

Alternatively, we can make *each* instance of `Foo` and `Stack` a friend of its *corresponding* instance of `Sample`:

```
template <class T>
class Sample {              // one-to-one friendship
    friend Foo<T>;
    friend Stack<T>;
    //...
};
```

This means that, for example, `Foo<int>` and `Stack<int>` are friends of `Sample<int>`, but not `Sample<double>`.

The extreme case of making *all* instances of `Foo` and `Stack` friends of *all* instances of `Sample` is expressed as:

```
template <class T>
class Sample {              // many-to-many friendship
    template <class T> friend Foo;
    template <class T> friend class Stack;
    //...
};
```

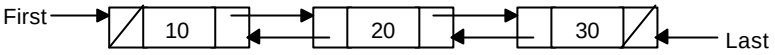
The choice as to which form of friendship to use depends on the intentions of the programmer.

□

Example: Doubly-linked Lists

A **container type** is a type which in turn contains objects of another type. A linked-list represents one of the simplest and most popular forms of container types. It consists of a set of elements, each of which contains a pointer to the next element in the list. In a doubly-linked list, each element also contains a pointer to the previous element in the list. Figure 9.1 illustrates a doubly-linked list of integers.

Figure 9.1 A doubly-linked list of integers.



Because a container class can conceivably contain objects of any type, it is best defined as a class template. Listing 9.6 show the definition of doubly-linked lists as two class templates.

Listing 9.6

```
 1  #include <iostream.h>

 2  enum  Bool {false, true};
 3  template <class Type> class List;    // forward declaration

 4  template <class Type>
 5  class ListElem {
 6  public:
 7                  ListElem    (const Type elem) : val(elem)
 8                              {prev = next = 0;}
 9      Type&       Value       (void)      {return val;}
10      ListElem*   Prev        (void)      {return prev;}
11      ListElem    *   Next        (void)      {return next;}
12      friend class List<Type>; // one-to-one friendship
13  protected:
14      Type        val;        // the element value
15      ListElem    *prev;      // previous element in the list
16      ListElem    *next;      // next element in the list
17  };
18  //--------------------------------------------------------
19  template <class Type>
20  class List {
21  public:
22                      List    (void)  {first = last = 0;}
23                      ~List   (void);
24      virtual void    Insert  (const Type&);
25      virtual void    Remove  (const Type&);
26      virtual Bool    Member  (const Type&);
27      friend  ostream& operator <<(ostream&, List&);
28  protected:
29      ListElem<Type>  *first; // first element in the list
30      ListElem<Type>  *last;  // last element in the list
31  };
```


Annotation

3 This forward declaration of the `List` class is necessary because `ListElem` refers to `List` before the latter's definition.

5-17 `ListElem` represents a list element. It consists of a value whose type is denoted by the type parameter `Type`, and two pointers which point to the previous and next elements in the list. `List` is declared as a one-to-one friend of `ListElem`, because the former's implementation requires access to the nonpublic members of the latter.

20 `List` represents a doubly-linked list.

24 `Insert` inserts a new element in front of the list.

25 `Remove` removes the list element, if any, whose value matches its parameter.

26 Member returns true if `val` is in the list, and false otherwise.

27 Operator `<<` is overloaded for the output of lists.

29-30 `First` and `last`, respectively, point to the first and last element in the list. Note that these two are declared of type `ListElem<Type>*` and not `ListElem*`, because the declaration is outside the `ListElem` class.

`Insert`, `Remove`, and `Element` are all defined as virtual to allow a class derived from `List` to override them.

All of the member functions of `ListElem` are defined inline. The definition of `List` member functions is as follows:

```
template <class Type>
List<Type>::~List (void)
{
    ListElem<Type> *handy;
    ListElem<Type> *next;

    for (handy = first; handy != 0; handy = next) {
       next = handy->next;
       delete handy;
    }
}
```

```
//------------------------------------------------------------------
-
template <class Type>
void List<Type>::Insert (const Type &elem)
{
    ListElem<Type> *handy = new ListElem<Type>(elem);

    handy->next = first;
    if (first != 0)
       first->prev = handy;
    if (last == 0)
       last = handy;
    first = handy;
}
//------------------------------------------------------------------
-
template <class Type>
void List<Type>::Remove (const Type &val)
{
    ListElem<Type> *handy;

    for (handy = first; handy != 0; handy = handy->next) {
       if (handy->val == val) {
           if (handy->next != 0)
               handy->next->prev = handy->prev;
           else
               last = handy->prev;
           if (handy->prev != 0)
               handy->prev->next = handy->next;
           else
               first = handy->next;
           delete handy;
       }
    }
}
//------------------------------------------------------------------
-
template <class Type>
Bool List<Type>::Member (const Type &val)
{
    ListElem<Type> *handy;

    for (handy = first; handy != 0; handy = handy->next)
       if (handy->val == val)
           return true;
    return false;
}
```

The `<<` is overloaded for both classes. The overloading of `<<` for `ListElem` does not require it to be declared a friend of the class because it is defined in terms of public members only:

```
template <class Type>
ostream& operator << (ostream &os, ListElem<Type> &elem)
{
    os << elem.Value();
    return os;
}
//-----------------------------------------------------------------
-
template <class Type>
ostream& operator << (ostream &os, List<Type> &list)
{
    ListElem<Type> *handy = list.first;

    os << "< ";
    for (; handy != 0; handy = handy->Next())
       os << *handy << ' ';
    os << '>';
    return os;
}
```

Here is a simple test of the class which creates the list shown in Figure 9.1:

```
int main (void)
{
    List<int>   list;

    list.Insert(30);
    list.Insert(20);
    list.Insert(10);
    cout << "list = " << list << '\n';
    if (list.Member(20)) cout << "20 is in list\n";
    cout << "Removed 20\n";
    list.Remove(20);
    cout << "list = " << list << '\n';
    return 0;
}
```

It will produce the following output:

```
list = < 10 20 30 >
20 is in list
Removed 20
< 10 30 >
```

□

Derived Class Templates

A class template (or its instantiation) can serve as the base of a derived class:

```
template <class Type>
class SmartList : public List<Type>;     // template base

class Primes : protected List<int>;      // instantiated base
```

A template base class, such as `List`, should always be accompanied with its parameter list (or arguments if instantiated). The following is therefore invalid:

```
template <class Type>
class SmartList : public List;       // illegal! <Type> missing
```

It would be equally incorrect to attempt to derive a nontemplate class from a (non-instantiated) template class:

```
class SmartList : public List<Type>; // illegal! template expected
```

It is, however, perfectly acceptable for a normal class to serve as the base of a derived template class:

```
class X;
template <class Type> class Y : X;       // ok
```

As an example of a derived class template, consider deriving a `Set` class from `List`. Given that a set consists of unique elements only (i.e., no repetitions), all we need to do is override the `Insert` member function to ensure this (see Listing 9.7).

Listing 9.7

```
1  template <class Type>
2  class Set : public List<Type> {
3  public:
4      virtual void    Insert  (const Type &val)
5                      {if (!Member(val)) List<Type>::Insert(val);}
6  };
```

□

Exercises

9.1 Define a `Swap` function template for swapping two objects of the same type.

9.2 Rewrite the `BubbleSort` function (Exercise 5.4) as a function template. Provide a specialization of the function for strings.

9.3 Rewrite the `BinaryTree` class (Exercise 6.6) as a class template. Provide a specialization of the class for strings.

9.4 Rewrite the `Database`, `BTree`, and `BStar` classes (Exercise 8.4) as class templates.

□

10. Exception Handling

An **exception** is a run-time error. Proper handling of exceptions is an important programming issue. This is because exceptions can and do happen in practice and programs are generally expected to behave gracefully in face of such exceptions. Unless an exception is properly handled, it is likely to result in abnormal program termination and potential loss of work. For example, an undetected division by zero or dereferencing of an invalid pointer will almost certainly terminate the program abruptly.

Exception handling consists of three things: (i) the detecting of a run-time error, (ii) raising an exception in response to the error, and (ii) taking corrective action. The latter is called **recovery**. Some exceptions can be fully recovered from so that execution can proceed unaffected. For example, an invalid argument value passed to a function may be handled by substituting a reasonable default value for it. Other exceptions can only be partially handled. For example, exhaustion of the heap memory can be handled by abandoning the current operation and returning to a state where other operations (such as saving the currently open files to avoid losing their contents) can be attempted.

C++ provides a language facility for the uniform handling of exceptions. Under this scheme, a section of code whose execution may lead to run-time errors is labeled as a **try block**. Any fragment of code activated during the execution of a try block can raise an exception using a **throw clause**. All exceptions are typed (i.e., each exception is denoted by an object of a specific type). A try block is followed by one or more **catch clauses**. Each catch clause is responsible for the handling of exceptions of a particular type.

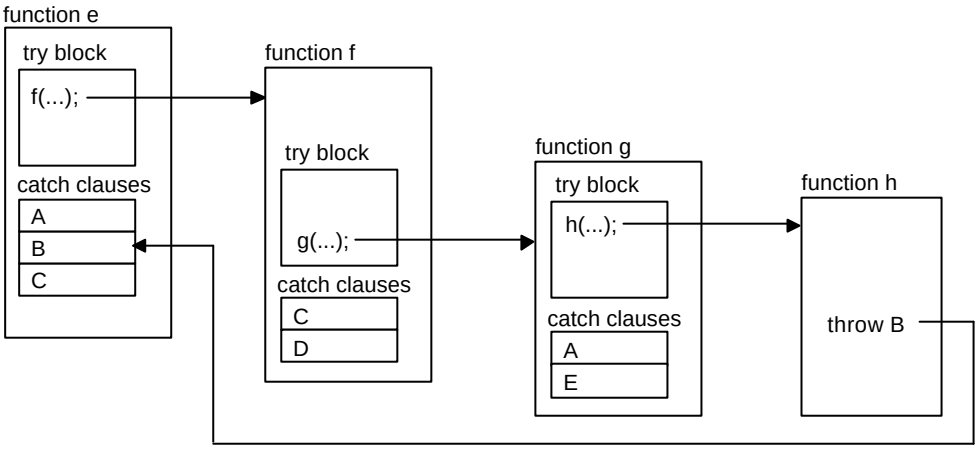
When an exception is raised, its type is compared against the catch clauses following it. If a matching clause is found then its handler is executed. Otherwise, the exception is propagated up, to an immediately enclosing try block (if any). The process is repeated until either the exception is handled by a matching catch clause or it is handled by a default handler.

Flow Control

Figure 10.1 illustrates the flow of control during exception handling. It shows a function *e* with a try block from which it calls *f*; *f* calls another function *g* from its own try block, which in turn calls *h*. Each of the try blocks is followed by a list of catch clauses. Function *h* throws an exception of type *B*. The enclosing try block's catch clauses are examined (i.e., *A* and *E*); neither matches *B*. The exception is therefore propagated to the catch clauses of the enclosing try block (i.e., *C* and *D*), which do not match *B* either. Propagating the exception further up, the catch clauses following the try block in *e* (i.e., *A*, *B*, and *C*) are examined next, resulting in a match.

At this point flow of control is transferred from where the exception was raised in *h* to the catch clause in *e*. The intervening stack frames for *h*, *g*, and *f* are unwound: all automatic objects created by these functions are properly destroyed by implicit calls to their destructors.

Figure 10.1 Flow control in exception handling.



Two points are worth noting. First, once an exception is raised and handled by a matching catch clause, the flow of control is *not* returned to where the exception was raised. The best that the program can do is to re-attempt the code that resulted in the exception (e.g., call *f* again in the above example). Second, the only role of a catch clause in life is to handle exceptions. If no exception is raised during the execution of a try block, then the catch clauses following it are simply ignored.

The Throw Clause

An exception is raised by a throw clause, which has the general form

```
throw object;
```

where *object* is an object of a built-in or user-defined type. Since an exception is matched by the type of *object* and not its value, it is customary to define classes for this exact purpose.

For example, recall the `Stack` class template discussed in Chapter 9 (see Listing 10.1).

Listing 10.1

```
template <class Type>
class Stack {
public:
            Stack   (int max);
            ~Stack  (void)          {delete [] stack;}
    void    Push        (Type &val);
    void    Pop     (void);
    Type&   Top     (void);
    friend  ostream& operator << (ostream&, Stack<Type>);
private:
    Type        *stack;
    int         top;
    const int   maxSize;
};
```

There are a number of potential run-time errors which may affect the member functions of `Stack`:

- The constructor parameter `max` may be given a nonsensical value. Also, the constructor's attempt at dynamically allocating storage for `stack` may fail due to heap exhaustion. We raise exceptions `BadSize` and `HeapFail` in response to these:

```
template <class Type>
Stack<Type>::Stack (int max) : maxSize(max)
{
    if (max <= 0)
        throw BadSize();
    if ((stack = new Type[max]) == 0)
        throw HeapFail();
    top = -1;
}
```

- An attempt to push onto a full stack results in an overflow. We raise an `Overflow` exception in response to this:


```
template <class Type>
void Stack<Type>::Push (Type &val)
{
    if (top+1 < maxSize)
        stack[++top] = val;
    else
        throw Overflow();
}
```

- An attempt to pop from an empty stack results in an underflow. We raise an `Underflow` exception in response to this:

```
template <class Type>
void Stack<Type>::Pop (void)
{
    if (top >= 0)
        --top;
    else
        throw Underflow();
}
```

- Attempting to examine the top element of an empty stack is clearly an error. We raise an `Empty` exception in response to this:

```
template <class Type>
Type &Stack<Type>::Top (void)
{
    if (top < 0)
        throw Empty();
    return stack[top];
}
```

Suppose that we have defined a class named `Error` for exception handling purposes. The above exceptions are easily defined as derivations of `Error`:

```
class Error         { /* ... */ };
class BadSize   : public Error {};
class HeapFail  : public Error {};
class Overflow  : public Error {};
class Underflow : public Error {};
class Empty     : public Error {};
```

The Try Block and Catch Clauses

A code fragment whose execution may potentially raise exceptions is enclosed by a try block, which has the general form

```
try {
    statements
}
```

where *statements* represents one or more semicolon-terminated statements. In other words, a try block is like a compound statement preceded by the `try` keyword.

A try block is followed by catch clauses for the exceptions which may be raised during the execution of the block. The role of the catch clauses is to handle the respective exceptions. A catch clause (also called a **handler**) has the general form

```
catch (type par)    { statements }
```

where *type* is the type of the object raised by the matching exception, *par* is optional and is an identifier bound to the object raised by the exception, and *statements* represents zero or more semicolon-terminated statements.

For example, continuing with our `Stack` class, we may write:

```
try {
    Stack<int> s(3);
    s.Push(10);
    //...
    s.Pop();
    //...
}
catch (Underflow)   {cout << "Stack underflow\n";}
catch (Overflow)        {cout << "Stack overflow\n";}
catch (HeapFail)        {cout << "Heap exhausted\n";}
catch (BadSize)     {cout << "Bad stack size\n";}
catch (Empty)       {cout << "Empty stack\n";}
```

For simplicity, the `catch` clauses here do nothing more than outputting a relevant message.

When an exception is raised by the code within the try block, the catch clauses are examined in the order they appear. The first matching catch clause is selected and its statements are executed. The remaining catch clauses are ignored.

A catch clause (of type *C*) matches an exception (of type *E*) if:

- *C* and *E* are the same type, or
- One is a reference or constant of the other type, or

- One is a nonprivate base class of the other type, or
- Both are pointers and one can be converted to another by implicit type conversion rules.

Because of the way the catch clauses are evaluated, their order of appearance is significant. Care should be taken to place the types which are likely to mask other types last. For example, the clause type `void*` will match any pointer and should therefore appear *after* other pointer type clauses:

```
try {
    //...
}
catch (char*)   {/*...*/}
catch (Point*)  {/*...*/}
catch (void*)   {/*...*/}
```

The special catch clause type

```
catch (...)     { /* ... */ }
```

will match any exception type and if used, like a default case in a switch statement, should always appear last.

The statements in a catch clause can also throw exceptions. The case where the matched exception is to be propagated up can be signified by an empty throw:

```
catch (char*)   {
    //...
    throw;              // propagate up the exception
}
```

An exception which is not matched by any catch clause after a try block, is propagated up to an enclosing try block. This process is continued until either the exception is matched or no more enclosing try block remains. The latter causes the predefined function `terminate` to be called, which simply terminates the program. This function has the following type:

```
typedef void (*TermFun)(void);
```

The default `terminate` function can be overridden by calling `set_terminate` and passing the replacing function as its argument:

```
TermFun set_terminate(TermFun);
```

`Set_terminate` returns the previous setting.

Function Throw Lists

It is a good programming practice to specify what exceptions a function may throw. This enables function users to quickly determine the list of exceptions that their code will have to handle. A function prototype may be appended with a throw list for this purpose:

type function (*parameters*) `throw` (*exceptions*);

where *exceptions* denotes a list of zero or more comma-separated exception types which *function* may directly or indirectly throw. The list is also an assurance that *function* will not throw any other exceptions.

For example,

```
void Encrypt (File &in, File &out, char *key)
     throw (InvalidKey, BadFile, const char*);
```

specifies that `Encrypt` may throw an `InvalidKey`, `BadFile`, or `const char*` exception, but none other. An empty throw list specifies that the function will not throw any exceptions:

```
void Sort (List list) throw ();
```

In absence of a throw list, the only way to find the exceptions that a function may throw is to study its code (including other functions that it calls). It is generally expected to at least define throw lists for frequently-used functions.

Should a function throw an exception which is not specified in its throw list, the predefined function `unexpected` is called. The default behavior of `unexpected` is to terminate the program. This can be overridden by calling `set_unexpected` (which has the same signature as `set_terminate`) and passing the replacing function as its argument:

```
TermFun set_unexpected(TermFun);
```

As before, `set_unexpected` returns the previous setting.

Exercises

10.1 Consider the following function which is used for receiving a packet in a network system:

```
void ReceivePacket (Packet *pack, Connection *c)
{
    switch (pack->Type()) {
        case controlPack:   //...
                            break;
        case dataPack:      //...
                            break;
        case diagnosePack:  //...
                            break;
        default:            //...
    }
}
```

Suppose we wish to check for the following errors in `ReceivePacket`:

- That connection `c` is active. `Connection::Active()` will return true if this is the case.
- That no errors have occurred in the transmission of the packet. `Packet::Valid()` will return true if this is the case.
- That the packet type is known (the `default` case is exercised otherwise).

Define suitable exceptions for the above and modify `ReceivePacket` so that it throws an appropriate exception when any of the above cases is not satisfied. Also define a throw list for the function.

10.2 Define appropriate exceptions for the `Matrix` class (see Chapter 7) and modify its functions so that they throw exceptions when errors occur, including the following:

- When the sizes of the operands of `+` and `-` are not identical.
- When the number of the columns of the first operand of `*` does not match the number of rows of its second operand.
- When the row or column specified for `()` is outside its range.
- When heap storage is exhausted.

11. The IO Library

C++ has no built-in Input/Output (IO) capability. Instead, this capability is provided by a library. The standard C++ IO library is called the **iostream** library. The definition of the library classes is divided into three header files. An additional header file defines a set of **manipulators** which act on streams. These are summarized by Table 11.1.

Figure 11.1 relates these header files to a class hierarchy for a UNIX-based implementation of the iostream class hierarchy. The highest-level classes appear unshaded. A user of the iostream library typically works with these classes only. Table 11.2 summarizes the role of these high-level classes. The library also provides four predefined stream objects for the common use of programs. These are summarized by Table 11.3.

Table 11.1 Iostream header files.

Header File	Description
`iostream.h`	Defines a hierarchy of classes for low-level (untyped character-level) IO and high-level (typed) IO. This includes the definition of the `ios`, `istream`, `ostream`, and `iostream` classes.
`fstream.h`	Derives a set of classes from those defined in `iostream.h` for file IO. This includes the definition of the `ifstream`, `ofstream`, and `fstream` classes.
`strstream.h`	Derives a set of classes from those defined in `iostream.h` for IO with respect to character arrays. This includes the definition of the `istrstream`, `ostrstream`, and `strstream` classes.
`iomanip.h`	Defines a set of manipulator which operate on streams to produce useful effects.

Table 11.2 Highest-level iostream classes.

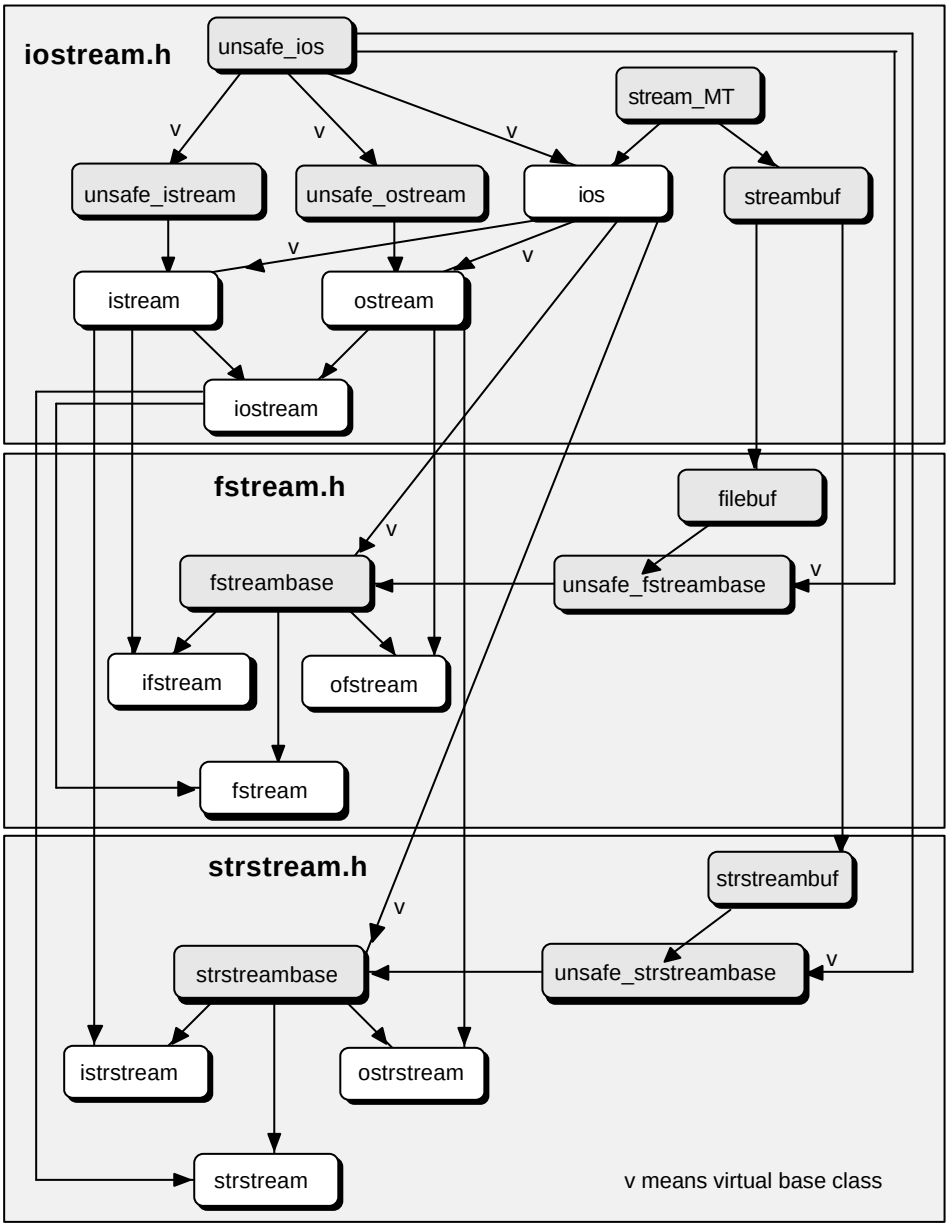
Form of IO	Input	Output	Input and Output
Standard IO	`istream`	`ostream`	`iostream`
File IO	`ifstream`	`ofstream`	`fstream`
Array of char IO	`istrstream`	`ostrstream`	`strstream`

Table 11.3 Predefined streams.

Stream	Type	Buffered	Description
`cin`	`istream`	Yes	Connected to standard input (e.g., the keyboard)
`cout`	`ostream`	Yes	Connected to standard output (e.g., the monitor)
`clog`	`ostream`	Yes	Connected to standard error (e.g., the monitor)
`cerr`	`ostream`	No	Connected to standard error (e.g., the monitor)

A stream may be used for input, output, or both. The act of reading data from an input stream is called **extraction**. It is performed using the >> operator (called the *extraction operator*) or an iostream member function. Similarly, the act of writing data to an output stream is called **insertion**, and is performed using the << operator (called the *insertion operator*) or an iostream member function. We therefore speak of 'extracting data from an input stream' and 'inserting data into an output stream'.

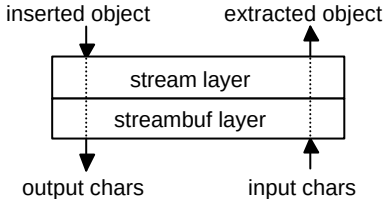
Figure 11.1 Iostream class hierarchy.



The Role of streambuf

The iostream library is based on a two layer model. The upper layer deals with formatted IO of typed objects (built-in or user-defined). The lower layer deals with unformatted IO of streams of characters, and is defined in terms of streambuf objects (see Figure 11.2). All stream classes contain a pointer to a streambuf object or one derived from it.

Figure 11.2 Two-layer IO model.



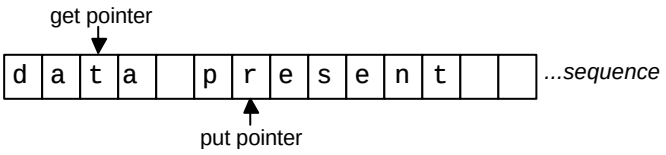
The streambuf layer provides buffering capability and hides the details of physical IO device handling. Under normal circumstances, the user need not worry about or directly work with streambuf objects. These are indirectly employed by streams. However, a basic understanding of how a streambuf operates makes it easier to understand some of the operations of streams.

Think of a streambuf as a sequence of characters which can grow or shrink. Depending on the type of the stream, one or two pointers are associated with this sequence (see Figure 11.3):

- A **put pointer** points to the position of the next character to be deposited into the sequence as a result of an insertion.
- A **get pointer** points to the position of the next character to be fetched from the sequence as a result of an extraction.

For example, `ostream` only has a put pointer, `istream` only has a get pointer, and `iostream` has both pointers.

Figure 11.3 Streambuf put and get pointers.



When a stream is created, a streambuf is associated with it. Therefore, the stream classes provide constructors which take a `streambuf*` argument. All stream classes overload the insertion and extraction operators for use with a `streambuf*` operand. The insertion or extraction of a streambuf causes the entire stream represented by it to be copied. □

Stream Output with ostream

Ostream provides formatted output capability. Use of the insertion operator << for stream output was introduced in Chapter 1, and employed throughout this book. The overloading of the insertion operator for user-defined types was discussed in Chapter 7. This section looks at the ostream member functions.

The `put` member function provides a simple method of inserting a single character into an output stream. For example, assuming that `os` is an ostream object,

```
os.put('a');
```

inserts `'a'` into `os`.

Similarly, `write` inserts a string of characters into an output stream. For example,

```
os.write(str, 10);
```

inserts the first 10 characters from `str` into `os`.

An output stream can be flushed by invoking its `flush` member function. Flushing causes any buffered data to be immediately sent to output:

```
os.flush();      // flushes the os buffer
```

The position of an output stream put pointer can be queried using `tellp` and adjusted using `seekp`. For example,

```
os.seekp(os.tellp() + 10);
```

moves the put pointer 10 characters forward. An optional second argument to `seekp` enables the position to be specified relatively rather than absolutely. For example, the above is equivalent to:

```
os.seekp(10, ios::cur);
```

The second argument may be one of:

- `ios::beg` for positions relative to the beginning of the stream,
- `ios::cur` for positions relative to the current put pointer position, or
- `ios::end` for positions relative to the end of the stream.

These are defined as a public enumeration in the `ios` class.

Table 11.4 summarizes the ostream member functions. All output functions with an `ostream&` return type, return the stream for which they are invoked. Multiple calls to such functions can be concatenated (i.e., combined into one statement). For example,

```
os.put('a').put('b');
```

is valid and is equivalent to:

```
os.put('a');
os.put('b');
```

Table 11.4 Member functions of ostream.

`ostream (streambuf*);` The constructor associates a streambuf (or its derivation) with the class to provide an output stream.
`ostream& put (char);` Inserts a character into the stream.

`ostream& write (const signed char*, int n);`
`ostream& write (const unsigned char*, int n);`
Inserts n signed or unsigned characters into the stream.

`ostream& flush ();` Flushes the stream.
`long tellp ();` Returns the current stream put pointer position.

`ostream& seekp (long, seek_dir = ios::beg);`
Moves the put pointer to a character position in the stream relative to the beginning, the current, or the end position:
`enum seek_dir {beg, cur, end};`

□

Stream Input with istream

Istream provides formatted input capability. Use of the extraction operator `>>` for stream input was introduced in Chapter 1. The overloading of the extraction operator for user-defined types was discussed in Chapter 7. This section looks at the istream member functions.

The `get` member function provides a simple method of extracting a single character from an input stream. For example, assuming that `is` is an istream object,

```
int ch = is.get();
```

extracts and returns the character denoted by the get pointer of `is`, and advances the get pointer. A variation of `get`, called `peek`, does the same but does not advance the get pointer. In other words, it allows you to examine the next input character without extracting it. The effect of a call to `get` can be canceled by calling `putback` which deposits the extracted character back into the stream:

```
is.putback(ch);
```

The return type of `get` and `peek` is an `int` (not `char`). This is because the end-of-file character (`EOF`) is usually given the value -1.

The behavior of `get` is different from the extraction operator in that the former does not skip blanks. For example, an input line consisting of

```
x y
```

(i.e., `'x'`, space, `'y'`, newline) would be extracted by four calls to `get`. the same line would be extracted by two applications of `>>`.

Other variations of `get` are also provided. See Table 11.5 for a summary.

The `read` member function extracts a string of characters from an input stream. For example,

```
char buf[64];
is.read(buf, 64);
```

extracts up to 64 characters from `is` and deposits them into `buf`. Of course, if `EOF` is encountered in the process, less characters will be extracted. The actual number of characters extracted is obtained by calling `gcount`.

A variation of `read`, called `getline`, allows extraction of characters until a user-specified delimiter is encountered. For example,

```
is.getline(buf, 64, '\t');
```

is similar to the above call to `read` but stops the extraction if a tab character is encountered. The delimiter, although extracted if encountered within the specified number of characters, is not deposited into `buf`.

Input characters can be skipped by calling `ignore`. For example,

```
is.ignore(10, '\n');
```

extracts and discards up to 10 characters but stops if a newline character is encountered. The delimiters itself is also extracted and discarded.

The position of an input stream get pointer can be queried using `tellg` and adjusted using `seekg`. For example,

```
is.seekp(is.tellg() - 10);
```

moves the get pointer 10 characters backward. An optional second argument to `seekg` enables the position to be specified relatively rather than absolutely. For example, the above is equivalent to:

```
is.seekg(-10, ios::cur);
```

As with `seekp`, the second argument may be one of `ios::beg`, `ios::cur`, or `ios::end`.

Table 11.5 summarizes the istream member functions. All input functions with an `istream&` return type, return the stream for which they are invoked. Multiple calls to such functions can therefore be concatenated. For example,

```
is.get(ch1).get(ch2);
```

is valid and is equivalent to:

```
is.get(ch1);
is.get(ch2);
```

The `iostream` class is derived from the `istream` and `ostream` classes and inherits their public members as its own public members:

```
class iostream : public istream, public ostream {
    //...
};
```

An iostream object is used for both insertion and extraction; it can invoke any of the functions listed in Tables 11.4 and 11.5.

Table 11.5 Member functions of istream.

`istream (streambuf*)` The constructor associates a streambuf (or its derivation) with the class to provide an input stream.

`int get ();`
`istream& get (signed char&);`
`istream& get (unsigned char&);`
`istream& get (streambuf&, char = '\n');`
The first version extracts the next character (including `EOF`). The second and third versions are similar but instead deposit the character into their parameter. The last version extracts and deposit characters into the given streambuf until the delimiter denoted by its last parameter is encountered.

`int peek ();` Returns the next input character without extracting it.
`istream& putback (char);` Pushes an extracted character back into the stream.

`istream& read (signed char*, int n);`
`istream& read (unsigned char*, int n);`
Extracts up to `n` characters into the given array, but stops if `EOF` is encountered.

`istream& getline (signed char*, int n, char = '\n');`
`istream& getline (unsigned char*, int n, char = '\n');`
Extracts at most `n-1` characters, or until the delimiter denoted by the last parameter or `EOF` is encountered, and deposit them into the given array, which is always null-terminated. The delimiter, if encountered and extracted, is not deposited into the array.

`int gcount ();` Returns the number of characters last extracted as a result of calling `read` or `getline`.
`istream& ignore (int n = 1, int = EOF);` Skips up to `n` characters, but extracts and stops if the delimiter denoted by the last parameter is encountered.
`long tellg ();` Returns the current stream get pointer position.

`istream& seekg (long, seek_dir = ios::cur);`
Moves the get pointer to a character position in the stream relative to the beginning, the current, or the end position:
`enum seek_dir {beg, cur, end};`

□

Using the ios Class

Ios provides capabilities common to both input and output streams. It uses a streambuf for buffering of data and maintains operational information on the state of the streambuf (i.e., IO errors). It also keeps formatting information for the use of its client classes (e.g., istream and ostream).

The definition of ios contains a number of public enumerations whose values are summarized by Table 11.6. The `io_state` values are used for the `state` data member which is a bit vector of IO error flags. The formatting flags are used for the `x_flags` data member (a bit vector). The `open_mode` values are bit flags for specifying the opening mode of a stream. The `seek_dir` values specify the seek direction for `seekp` and `seekg`.

Table 11.6 Useful public enumerations in ios.

`enum io_state`:	Provides status flags (for `ios::state`).
`ios::goodbit`	When `state` is set to this value, it means that all is ok.
`ios::eofbit`	End-of-file has been reached.
`ios::badbit`	An invalid operation has been attempted.
`ios::failbit`	The last IO operation attempted has failed.
`ios::hardfail`	An unrecoverable error has taken place.
Anonymous `enum`:	Provides formatting flags.
`ios::left`	Left-adjust the output.
`ios::right`	Right-adjust the output.
`ios::internal`	Output padding indicator.
`ios::dec`	Convert to decimal.
`ios::oct`	Convert to octal.
`ios::hex`	Convert to hexadecimal.
`ios::showbase`	Show the base on output.
`ios::showpoint`	Show the decimal point on output.
`ios::uppercase`	Use upper case for hexadecimal output.
`ios::showpos`	Show the + symbol for positive integers.
`ios::fixed`	Use the floating notation for reals.
`ios::scientifi` `c`	Use the scientific notation for reals.
`ios::skipws`	Skip blanks (white spaces) on input.
`ios::unitbuf`	Flush all streams after insertion.
`enum open_mode`:	Provides values for stream opening mode.
`ios::in`	Stream open for input.
`ios::out`	Stream open for output.
`ios::app`	Append data to the end of the file.
`ios::ate`	Upon opening the stream, seek to EOF.
`ios::trunc`	Truncate existing file.
`ios::noreplace`	Open should fail if file already exists.
`ios::nocreate`	Open should fail if file does not already exist.
`ios::binary`	Binary file (as opposed to default text file).
`enum seek_dir`:	Provides values for relative seek.
`ios::beg`	Seek relative to the beginning of the stream.
`ios::cur`	Seek relative to the current put/get pointer position.
`ios::end`	Seek relative to the end of the stream.

IO operations may result in IO errors, which can be checked for using a number of ios member functions. For example, `good` returns nonzero if no error has occurred:

```
if (s.good())
    // all is ok...
```

where `s` is an iostream. Similarly, `bad` returns nonzero if an invalid IO operation has been attempted:

```
if (s.bad())
    // invalid IO operation...
```

and `fail` returns true if the last attempted IO operation has failed (or if `bad()` is true):

```
if (s.fail())
    // last IO operation failed...
```

A shorthand for this is provided, based on the overloading of the `!` operator:

```
if (!s)     // same as: if (s.fail())
    // ...
```

The opposite shorthand is provided through the overloading of the `void*` so that it returns zero when `fail` returns nonzero. This makes it possible to check for errors in the following fashion:

```
if (cin >> str)
    // no error occurred
```

The entire error bit vector can be obtained by calling `rdstate`, and cleared by calling `clear`. User-defined IO operations can report errors by calling `setstate`. For example,

```
s.setstate(ios::eofbit | ios::badbit);
```

sets the `eofbit` and `badbit` flags.

Ios also provides various formatting member functions. For example, `precision` can be used to change the precision for displaying floating point numbers:

```
cout.precision(4);
cout << 233.123456789 << '\n';
```

This will produce the output:

```
233.1235
```

The `width` member function is used to specify the minimum width of the next output object. For example,

```
cout.width(5);
cout << 10 << '\n';
```

will use exactly 5 character to display 10:

```
   10
```

An object requiring more than the specified width will not be restricted to it. Also, the specified width applies only to the *next* object to be output. By default, spaces are used to pad the object up to the specified minimum size. The padding character can be changed using `fill`. For example,

```
cout.width(5);
cout.fill('*');
cout << 10 << '\n';
```

will produce:

```
***10
```

The formatting flags listed in Table 11.6 can be manipulated using the `setf` member function. For example,

```
cout.setf(ios::scientific);
cout << 3.14 << '\n';
```

will display:

```
3.14e+00
```

Another version of `setf` takes a second argument which specifies formatting flags which need to be reset beforehand. The second argument is typically one of:

```
ios::basefield   ≡ ios::dec | ios::oct | ios::hex
ios::adjustfield ≡ ios::left | ios::right | ios::internal
ios::floatfield  ≡ ios::scientific | ios::fixed
```

For example,

```
cout.setf(ios::hex | ios::uppercase, ios::basefield);
cout << 123456 << '\n';
```

will display:

```
1E240
```


Formatting flags can be reset by calling `unsetf`, and set as a whole or examined by calling `flags`. For example, to disable the skipping of leading blanks for an input stream such as `cin`, we can write:

```
cin.unsetf(ios::skipws);
```

Table 11.7 summarizes the member functions of ios.

Table 11.7 Member functions of ios.

`ios (streambuf*);` The constructor associates a streambuf (or its derivation) with the class.
`void init (streambuf*);` Associates the specified streambuf with the stream.
`streambuf* rdbuf (void);` Returns a pointer to the stream's associated streambuf.
`int good (void);` Examines `ios::state` and returns zero if bits have been set as a result of an error.
`int bad (void);` Examines the `ios::badbit` and `ios::hardfail` bits in `ios::state` and returns nonzero if an IO error has occurred.
`int fail (void);` Examines the `ios::failbit`, `ios::badbit`, and `ios::hardfail` bits in `ios::state` and returns nonzero if an operation has failed.
`int eof (void);` Examines the `ios::eofbit` in `ios::state` and returns nonzero if the end-of-file has been reached.
`void clear (int = 0);` Sets the `ios::state` value to the value specified by the parameter.
`void setstate (int);` Sets the `ios::state` bits specified by the parameter.
`int rdstate (void);` Returns `ios::state`.

`int precision (void);`
`int precision (int);`
The first version returns the current floating-point precision. The second version sets the floating-point precision and returns the previous floating-point precision.

`int width (void);`
`int width (int);`
The first version returns the current field width. The second version sets the field width and returns the previous setting.

`char fill (void);`
`char fill (char);`
The first version returns the current fill character. The second version changes the fill character and returns the previous fill character.

`long setf (long);`
`long setf (long, long);`
The first version sets the formatting flags denoted by the parameter. The second version also clears the flags denoted by its second argument. Both return the previous setting.

`long unsetf (long);` Clears the formatting flags denoted by its parameter, and returns the previous setting.

`long flags (void);`
`long flags (long);`
The first version returns the format flags (this is a sequence of formatting bits). The second version sets the formatting flags to a given value (`flags(0)` restores default formats), and return the previous setting.

`ostream* tie (void);`
`ostream* tie (ostream*);`
Returns the tied stream, if any, and zero otherwise. The second version ties the stream denoted by its parameter to this stream and returns the previously-tied stream. When two streams are tied the use of one affects the other. For example, because `cin`, `cerr`, and `clog` are all tied to `cout`, using any of the first three causes `cout` to be flushed first.

□

Stream Manipulators

A manipulator is an identifier that can be inserted into an output stream or extracted from an input stream in order to produce a desired effect. For example, `endl` is a commonly-used manipulator which inserts a newline into an output stream and flushes it. Therefore,

```
cout << 10 << endl;
```

has the same effect as:

```
cout << 10 << '\n';
```

In general, most formatting operations are more easily expressed using manipulators than using `setf`. For example,

```
cout << oct << 10 << endl;
```

is an easier way of saying:

```
cout.setf(ios::oct, ios::basefield);
cout << 10 << endl;
```

Some manipulators also take parameters. For example, the `setw` manipulator is used to set the field width of the next IO object:

```
cout << setw(8) << 10;  // sets the width of 10 to 8 characters
```

Table 11.8 summarizes the predefined manipulators of the iostream library.

Table 11.8 Predefined manipulators.

Manipulator	Stream Type	Description
`endl`	output	Inserts a newline character and flushes the stream.
`ends`	output	Inserts a null-terminating character.
`flush`	output	Flushes the output stream.
`dec`	input/output	Sets the conversion base to decimal.
`hex`	input/output	Sets the conversion base to hexadecimal.
`oct`	input/output	Sets the conversion base to octal.
`ws`	input	Extracts blanks (white space) characters.
`setbase(int)`	input/output	Sets the conversion base to one of 8, 10, or 16.
`resetiosflags(long)`	input/output	Clears the status flags denoted by the argument.
`setiosflags(long)`	input/output	Sets the status flags denoted by the argument.
`setfill(int)`	input/output	Sets the padding character to the argument.
`setprecision(int)`	input/output	Sets the floating-point precision to the argument.
`setw(int)`	input/output	Sets the field width to the argument.

□

File IO with fstreams

A program which performs IO with respect to an external file should include the header file `fstream.h`. Because the classes defined in this file are derived from iostream classes, `fstream.h` also includes `iostream.h`.

A file can be opened for output by creating an ofstream object and specifying the file name and mode as arguments to the constructor. For example,

```
ofstream log("log.dat", ios::out);
```

opens a file named `log.dat` for output (see Table 11.6 for a list of the open mode values) and connects it to the ofstream `log`. It is also possible to create an ofstream object first and then connect the file later by calling `open`:

```
ofstream log;
log.open("log.dat", ios::out);
```

Because ofstream is derived from ostream, all the public member functions of the latter can also be invoked for ofstream objects. First, however, we should check that the file is opened as expected:

```
if (!log)
    cerr << "can't open 'log.dat'\n";
else {
    char *str = "A piece of text";
    log.write(str, strlen(str));
    log << endl;
}
```

The external file connected to an ostream can be closed and disconnected by calling `close`:

```
log.close();
```

A file can be opened for input by creating an ifstream object. For example,

```
ifstream inf("names.dat", ios::in);
```

opens the file `names.dat` for input and connects it to the ifstream `inf`. Because ifstream is derived from istream, all the public member functions of the latter can also be invoked for ifstream objects.

The fstream class is derived from iostream and can be used for opening a file for input as well as output. For example:

```
fstream iof;

iof.open("names.dat", ios::out);     // output
iof << "Adam\n";
```



```
iof.close();

char name[64];
iof.open("names.dat", ios::in);      // input
iof >> name;
iof.close();
```

Table 11.9 summarizes the member functions of ofstream, istream, and fstream (in addition to those inherited from their base classes).

Table 11.9 Member functions of ofstream, ifstream, and fstream.

`ofstream (void);`
`ofstream (int fd);`
`ofstream (int fd, char* buf, int size);`
`ofstream (const char*, int=ios::out, int=filebuf::openprot);`
The first version makes an ofstream which is not attached to a file. The second version makes an ofstream and connects it to an open file descriptor. The third version does the same but also uses a user-specified buffer of a given size. The last version makes an ofstream and opens and connects a specified file to it for writing.

`ifstream (void);`
`ifstream (int fd);`
`ifstream (int fd, char* buf, int size);`
`ifstream (const char*, int=ios::in, int=filebuf::openprot);`
Similar to ofstream constructors.

`fstream (void);`
`fstream (int fd);`
`fstream (int fd, char* buf, int size);`
`fstream (const char*, int, int=filebuf::openprot);`
Similar to ofstream constructors.

`void open (const char*, int, int = filebuf::openprot);` Opens a file for an ofstream, ifstream, or fstream.
`void close (void);` Closes the associated filebuf and file.
`void attach(int);` Connects to an open file descriptor.
`void setbuf(char*, int);` Assigns a user-specified buffer to the filebuf.
`filebuf* rdbuf (void);` Returns the associated filebuf.

□

Array IO with strstreams

The classes defined in `strstream.h` support IO operations with respect to arrays of characters. Insertion and extraction on such streams causes the data to be moved into and out of its character array. Because these classes are derived from iostream classes, this file also includes `iostream.h`.

The three highest-level array IO classes (ostrstream, istrstream, strstream) are very similar to the file IO counterparts (ofstream, ifstream, fstream). As before, they are derived from iostream classes and therefore inherit their member functions.

An ostrstream object is used for output. It can be created with either a dynamically-allocated internal buffer, or a user-specified buffer:

```
ostrstream odyn;                // dynamic buffer
char buffer[1024];
ostrstream ssta(buffer, 1024);  // user-specified buffer
```

The static version (`ssta`) is more appropriate for situations where the user is certain of an upper bound on the stream buffer size. In the dynamic version, the object is responsible for resizing the buffer as needed.

After all the insertions into an ostrstream have been completed, the user can obtain a pointer to the stream buffer by calling `str`:

```
char *buf = odyn.str();
```

This freezes `odyn` (disabling all future insertions). If `str` is not called before `odyn` goes out of scope, the class destructor will destroy the buffer. However, when `str` is called, this responsibility rests with the user. Therefore, the user should make sure that when `buf` is no longer needed it is deleted:

```
delete buf;
```

An istrstream object is used for input. Its definition requires a character array to be provided as a source of input:

```
char data[128];
//...
istrstream istr(data, 128);
```

Alternatively, the user may choose not to specify the size of the character array:

```
istrstream istr(data);
```

The advantage of the former is that extraction operations will not attempt to go beyond the end of `data` array.

Table 11.10 summarizes the member functions of ostrstream, istrstream, and strstream (in addition to those inherited from their base classes).

Table 11.10 Member functions of ostrstream, istrstream, and strstream.

`ostrstream (void);`
`ostrstream (char *buf, int size, int mode = ios::out);`
The first version creates an ostrstream with a dynamically-allocated buffer. The second version creates an ostrstream with a user-specified buffer of a given size.

`istrstream (const char *);`
`istrstream (const char *, int n);`
The first version creates an istrstream using a given string. The second version creates an istrstream using the first n bytes of a given string.

`strstream (void);`
`strstream (char *buf, int size, int mode);`
Similar to ostrstream constructors.

`char* pcount (void);` Returns the number of bytes currently stored in the buffer of an output stream.
`char* str (void);` Freezes and returns the output stream buffer which, if dynamically allocated, should eventually be deallocated by the user.
`strstreambuf* rdbuf (void);` Returns a pointer to the associated buffer.

□

Example: Program Annotation

Suppose we are using a language compiler which generates error message of the form:

```
Error 21, invalid expression
```

where 21 is the number of the line in the program file where the error has occurred. We would like to write a tool which takes the output of the compiler and uses it to annotate the lines in the program file which are reported to contain errors, so that, for example, instead of the above we would have something like:

```
0021    x = x * y +;
     Error: invalid expression
```

Listing 11.1 provides a function which performs the proposed annotation.

Annotation

6 `Annotate` takes two argument: `inProg` denotes the program file name and `inData` denotes the name of the file which contains the messages generated by the compiler.

8-9 `InProg` and `inData` are, respectively, connected to istreams `prog` and `data`.

12 `Line` is defined to be an `istrstream` which extracts from `dLine`.

21 Each time round this loop, a line of text is extracted from `data` into `dLine`, and then processed.

22-26 We are only interested in lines which start with the word `Error`. When a match is found, we reset the get pointer of `data` back to the beginning of the stream, ignore characters up to the space character before the line number, extract the line number into `lineNo`, and then ignore the remaining characters up to the comma following the line number (i.e., where the actual error message starts).

27-29 This loop skips `prog` lines until the line denoted by the error message is reached.

30-33 These insertions display the `prog` line containing the error and its annotation. Note that as a result of the re-arrangements, the line number is effectively removed from the error message and displayed next to the program line.

36-37 The ifstreams are closed before the function returning.

Listing 11.1

```
 1  #include <fstream.h>
 2  #include <strstream.h>
 3  #include <iomanip.h>
 4  #include <string.h>

 5  const int lineSize = 128;

 6  int Annotate (const char *inProg, const char *inData)
 7  {
 8      ifstream    prog(inProg, ios::in);
 9      ifstream    data(inData, ios::in);
10      char        pLine[lineSize];        // for prog lines
11      char        dLine[lineSize];        // for data lines
12      istrstream  line(dLine, lineSize);
13      char        *prefix = "Error";
14      int         prefixLen = strlen(prefix);
15      int         progLine = 0;
16      int         lineNo;

17      if (!prog || !data) {
18          cerr << "Can't open input files\n";
19          return -1;
20      }

21      while (data.getline(dLine, lineSize, '\n')) {
22          if (strncmp(dLine, prefix, prefixLen) == 0) {
23              line.seekg(0);
24              line.ignore(lineSize, ' ');
25              line >> lineNo;
26              line.ignore(lineSize, ',');

27              while (progLine < lineNo &&
28                     prog.getline(pLine, lineSize))
29                  ++progLine;
30              cout << setw(4) << setfill('0') << progLine
31                   << " " << pLine << endl;
32              cout << "     " << prefix << ":"
33                   << dLine + line.tellg() << endl;
34          }
35      }
36      prog.close();
37      data.close();
38      return 0;
39  }
```

The following `main` function provides a simple test for `Annotate`:

```
int main (void)
{
    return Annotate("prog.dat", "data.dat");
}
```

The contents of these two files are as follows:

prog.dat:

```
#defone size 100

main (void)
{
    integer n = 0;

    while (n < 10]
        ++n;
    return 0;
}
```

data.dat:

```
Error 1, Unknown directive: defone
Note 3, Return type of main assumed int
Error 5, unknown type: integer
Error 7, ) expected
```

When run, the program will produce the following output:

```
0001 #defone size 100
     Error: Unknown directive: defone
0005    integer n = 0;
     Error: unknown type: integer
0007    while (n < 10]
     Error: ) expected
```

□

Exercises

11.1 Use the istream member functions to define an overloaded version of the `>>` operator for the `Set` class (see Chapter 7) so that it can input sets expressed in the conventional mathematical notation (e.g., {2, 5, 1}).

11.2 Write a program which copies its standard input, line by line, to its standard output.

11.3 Write a program which copies a user-specified file to another user-specified file. Your program should be able to copy text as well as binary files.

11.4 Write a program which reads a C++ source file and checks that all instances of brackets are balanced, that is, each '(' has a matching ')', and similarly for [] and {}, except for when they appear inside comments or strings. A line which contains an unbalanced bracket should be reported by a message such as the following sent to standard output:

```
'{' on line 15 has no matching '}'
```

□

12. The Preprocessor

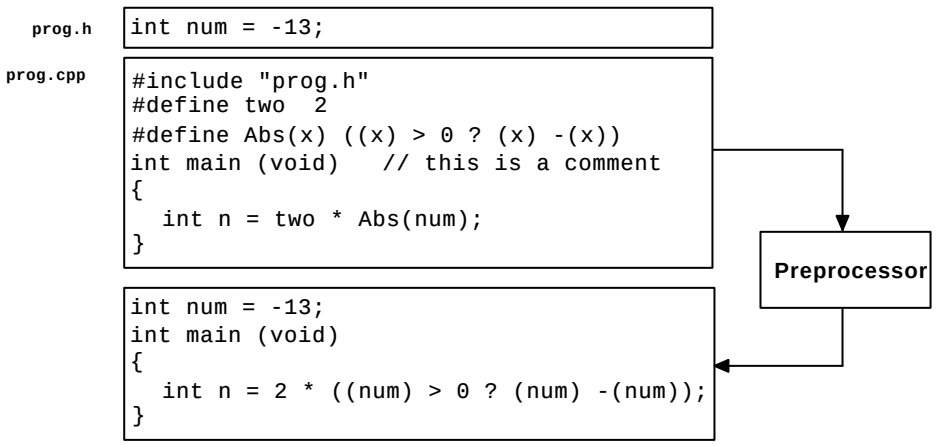
Prior to compiling a program source file, the C++ compiler passes the file through a preprocessor. The role of the preprocessor is to transform the source file into an equivalent file by performing the preprocessing instructions contained by it. These instructions facilitate a number of features, such as: file inclusion, conditional compilation, and macro substitution.

Figure 12.1 illustrates the effect of the preprocessor on a simple file. It shows the preprocessor performing the following:

- Removing program comments by substituting a single white space for each comment.
- Performing the file inclusion (`#include`) and conditional compilation (`#ifdef`, etc.) commands as it encounters them.
- 'Learning' the macros introduced by `#define`. It compares these names against the identifiers in the program, and does a substitution when it finds a match.

The preprocessor performs very minimal error checking of the preprocessing instructions. Because it operates at a text level, it is unable to check for any sort of language-level syntax errors. This function is performed by the compiler.

Figure 12.1 The role of the preprocessor.



Preprocessor Directives

Programmer instructions to the preprocessor (called **directives**) take the general form:

 # directive tokens

The # symbol should be the first non-blank character on the line (i.e., only spaces and tabs may appear before it). Blank symbols may also appear between the # and *directive*. The following are therefore all valid and have exactly the same effect:

```
#define size        100
    #define size        100
    #   define size    100
```

A directive usually occupies a single line. A line whose last non-blank character is \, is assumed to continue on the line following it, thus making it possible to define multiple line directives. For example, the following multiple line and single line directives have exactly the same effect:

```
#define CheckError  \
    if (error)      \
        exit(1)

#define CheckError  if (error)  exit(1)
```

A directive line may also contain comment; these are simply ignored by the preprocessor. A # appearing on a line on its own is simply ignored.

Table 12.1 summarizes the preprocessor directives, which are explained in detail in subsequent sections. Most directives are followed by one or more tokens. A token is anything other than a blank.

Table 12.1 Preprocessor directives.

Directive	Explanation
`#define`	Defines a macro
`#undef`	Undefines a macro
`#include`	Textually includes the contents of a file
`#ifdef`	Makes compilation of code conditional on a macro being defined
`#ifndef`	Makes compilation of code conditional on a macro not being defined
`#endif`	Marks the end of a conditional compilation block
`#if`	Makes compilation of code conditional on an expression being nonzero
`#else`	Specifies an else part for a `#ifdef`, `#ifndef`, or `#if` directive
`#elif`	Combination of `#else` and `#if`
`#line`	Change current line number and file name
`#error`	Outputs an error message
`#pragma`	Is implementation-specific

□

Macro Definition

Macros are defined using the `#define` directive, which takes two forms: plain and parameterized. A **plain macro** has the general form:

`#define` *identifier tokens*

It instructs the preprocessor to substitute *tokens* for every occurrence of *identifier* in the rest of the file (except for inside strings). The substitution *tokens* can be anything, even empty (which has the effect of removing *identifier* from the rest of the file).

Plain macros are used for defining symbolic constants. For example:

```
#define size    512
#define word        long
#define bytes   sizeof(word)
```

Because macro substitution is also applied to directive lines, an identifier defined by one macro can be used in a subsequent macro (e.g., use of `word` in `bytes` above). Given the above definitions, the code fragment

```
word n = size * bytes;
```

is macro-expanded to:

```
long n = 512 * sizeof(long);
```

Use of macros for defining symbolic constants has its origins in C, which had no language facility for defining constants. In C++, macros are less often used for this purpose, because `const`s can be used instead, with the added benefit of proper type checking.

A **parameterized macro** has the general form

`#define` *identifier*`(`*parameters*`)` *tokens*

where *parameters* is a list of one or more comma-separated identifiers. There should be no blanks between the *identifier* and `(`. Otherwise, the whole thing is interpreted as a plain macro whose substitution tokens part starts from `(`. For example,

```
#define Max(x,y)    ((x) > (y) ? (x) : (y))
```

defines a parameterized macro for working out the maximum of two quantities.

A parameterized macro is matched against a call to it, which is syntactically very similar to a function call. A call must provide a matching number of arguments.

As before, the *tokens* part of the macro is substituted for the call. Additionally, every occurrence of a parameter in the substituted *tokens* is substituted by the corresponding argument. This is called **macro expansion**. For example, the call

```
n = Max (n - 2, k +6);
```

is macro-expanded to:

```
n = (n - 2) > (k + 6) ? (n - 2) : (k + 6);
```

Note that the `(` in a macro call may be separated from the macro identifier by blanks.

It is generally a good idea to place additional brackets around each occurrence of a parameter in the substitution *tokens* (as we have done for `Max`). This protects the macro against undesirable operator precedence effects after macro expansion.

Overlooking the fundamental difference between macros and functions can lead to subtle programming errors. Because macros work at a textual level, the semantics of macro expansion is not necessarily equivalent to function call. For example, the macro call

```
Max(++i, j)
```

is expanded to

```
((++i) > (j) ? (++i) : (j))
```

which means that `i` may end up being incremented twice. Where as a function version of `Max` would ensure that `i` is only incremented once.

Two facilities of C++ make the use of parameterized macros less attractive than in C. First, C++ inline functions provide the same level of code efficiency as macros, without the semantics pitfalls of the latter. Second, C++ templates provide the same kind of flexibility as macros for defining generic functions and classes, with the added benefit of proper syntax analysis and type checking.

Macros can also be redefined. However, before a macro is redefined, it should be undefined using the `#undef` directive. For example:

```
#undef  size
#define size    128
#undef  Max
```

Use of `#undef` on an undefined identifier is harmless and has no effect.

□

Quote and Concatenation Operators

The preprocessor provides two special operators or manipulating macro parameters. The **quote operator** (#) is unary and takes a macro parameter operand. It transforms its operand into a string by putting double-quotes around it.

For example, consider a parameterized macro which checks for a pointer to be nonzero and outputs a warning message when it is zero:

```
#define CheckPtr(ptr)    \
        if ((ptr) == 0) cout << #ptr << " is zero!\n"
```

Use of the # operator allows the expression given as argument to `CheckPtr` to be literally printed as a part of the warning message. Therefore, the call

```
CheckPtr(tree->left);
```

is expanded as:

```
if ((tree->left) == 0) cout << "tree->left" << " is zero!\n";
```

Note that defining the macro as

```
#define CheckPtr(ptr)    \
        if ((ptr) == 0) cout << "ptr is zero!\n"
```

would not produce the desired effect, because macro substitution is not performed inside strings.

The concatenation operator (##) is binary and is used for concatenating two tokens. For example, given the definition

```
#define internal(var)    internal##var
```

the call

```
long    internal(str);
```

expands to:

```
long    internalstr;
```

This operator is rarely used for ordinary programs. It is very useful for writing translators and code generators, as it makes it easy to build an identifier out of fragments.

□

File Inclusion

A file can be textually included in another file using the `#include` directive. For example, placing

```
#include "constants.h"
```

inside a file *f* causes the contents of `contents.h` to be included in *f* in exactly the position where the directive appears. The included file is usually expected to reside in the same directory as the program file. Otherwise, a full or relative path to it should be specified. For example:

```
#include "../file.h"            // include from parent dir (UNIX)
#include "/usr/local/file.h"    // full path (UNIX)
#include "..\file.h"            // include from parent dir (DOS)
#include "\usr\local\file.h"    // full path (DOS)
```

When including system header files for standard libraries, the file name should be enclosed in <> instead of double-quotes. For example:

```
#include <iostream.h>
```

When the preprocessor encounters this, it looks for the file in one or more prespecified locations on the system (e.g., the directory `/usr/include/cpp` on a UNIX system). On most systems the exact locations to be searched can be specified by the user, either as an argument to the compilation command or as a system environment variable.

File inclusions can be nested. For example, if a file *f* includes another file *g* which in turn includes another file *h*, then effectively *f* also includes *h*.

Although the preprocessor does not care about the ending of an included file (i.e., whether it is `.h` or `.cpp` or `.cc`, etc.), it is customary to only include header files in other files.

Multiple inclusion of files may or may not lead to compilation problems. For example, if a header file contains only macros and declarations then the compiler will not object to their reappearance. But if it contains a variable definition, for example, the compiler will flag it as an error. The next section describes a way of avoiding multiple inclusions of the same file.

□

Conditional Compilation

The conditional compilation directives allow sections of code to be selectively included for or excluded from compilation, depending on programmer-specified conditions being satisfied. It is usually used as a portability tool for tailoring the program code to specific hardware and software architectures. Table 12.2 summarizes the general forms of these directives (*code* denotes zero or more lines of program text, and *expression* denotes a constant expression).

Table 12.2 General forms of conditional compilation directives.

Form	Explanation
`#ifdef` *identifier* *code* `#endif`	If *identifier* is a `#defined` symbol then *code* is included in the compilation process. Otherwise, it is excluded.
`#ifndef` *identifier* *code* `#endif`	If *identifier* is not a `#defined` symbol then *code* is included in the compilation process. Otherwise, it is excluded.
`#if` *expression* *code* `#endif`	If *expression* evaluates to nonzero then *code* is included in the compilation process. Otherwise, it is excluded.

`#ifdef` *identifier*
  code1
`#else`
  code2
`#endif`
	If *identifier* is a #defined symbol then *code1* is included in the compilation process and *code2* is excluded. Otherwise, *code2* is included and *code1* is excluded.
	Similarly, `#else` can be used with `#ifndef` and `#if`.

`#if` *expression1*
  code1
`#elif` *expression2*
  code2
`#else`
  code3
`#endif`
	If *expression1* evaluates to nonzero then only *code1* is included in the compilation process. Otherwise, if *expression2* evaluates to nonzero then only *code2* is included. Otherwise, *code3* is included.
	As before, the `#else` part is optional. Also, any number of `#elif` directives may appear after a `#if` directive.

Here are two simple examples:

```
// Different application start-ups for beta and final version:
#ifdef BETA
    DisplayBetaDialog();
#else
    CheckRegistration();
#endif

// Ensure Unit is at least 4 bytes wide:
#if sizeof(int) >= 4
    typedef int   Unit;
#elif sizeof(long) >= 4
    typedef long  Unit;
#else
    typedef char      Unit[4];
#endif
```


One of the common uses of `#if` is for temporarily omitting code. This is often done during testing and debugging when the programmer is experimenting with suspected areas of code. Although code may also be omitted by commenting its out (i.e., placing `/*` and `*/` around it), this approach does not work if the code already contains `/*...*/` style comments, because such comments cannot be nested.

Code is omitted by giving `#if` an expression which always evaluates to zero:

```
#if 0
    ...code to be omitted
#endif
```

The preprocessor provides an operator called `defined` for use is expression arguments of `#if` and `#elif`. For example,

```
#if defined BETA
```

has the same effect as:

```
#ifdef BETA
```

However, use of `defined` makes it possible to write compound logical expressions. For example:

```
#if defined ALPHA || defined BETA
```

Conditional compilation directives can be used to avoid the multiple of inclusion of files. For example, given an include file called `file.h`, we can avoid multiple inclusions of `file.h` in any other file by adding the following to `file.h`:

```
#ifndef _file_h_
#define _file_h_
      contents of file.h goes here
#endif
```

When the preprocessor reads the first inclusion of `file.h`, the symbol `_file_h_` is undefined, hence the contents is included, causing the symbol to be defined. Subsequent inclusions have no effect because the `#ifndef` directive causes the contents to be excluded.

□

Other Directives

The preprocessor provides three other, less-frequently-used directives. The `#line` directive is used to change the current line number and file name. It has the general form:

`#line` *number file*

where *file* is optional. For example,

```
#line 20 "file.h"
```

makes the compiler believe that the current line number is 20 and the current file name is `file.h`. The change remains effective until another `#line` directive is encountered. The directive is useful for translators which generate C++ code. It allows the line numbers and file name to be made consistent with the original input file, instead of any intermediate C++ file.

The `#error` directive is used for reporting errors by the preprocessor. It has the general form

`#error` *error*

where *error* may be any sequence of tokens. When the preprocessor encounters this, it outputs *error* and causes compilation to be aborted. It should therefore be only used for reporting errors which make further compilation pointless or impossible. For example:

```
#ifndef UNIX
#error This software requires the UNIX OS.
#endif
```

The `#pragma` directive is implementation-dependent. It is used by compiler vendors to introduce nonstandard preprocessor features, specific to their own implementation. Examples from the SUN C++ compiler include:

```
// align name and val starting addresses to multiples of 8 bytes:
#pragma align 8 (name, val)
char   name[9];
double val;

// call MyFunction at the beginning of program execution:
#pragma init (MyFunction)
```

□

Predefined Identifiers

The preprocessor provides a small set of predefined identifiers which denote useful information. The standard ones are summarized by Table 12.3. Most implementations augment this list with many nonstandard predefined identifiers.

Table 12.3 Standard predefined identifiers.

Identifier	Denotes
`__FILE__`	Name of the file being processed
`__LINE__`	Current line number of the file being processed
`__DATE__`	Current date as a string (e.g., "25 Dec 1995")
`__TIME__`	Current time as a string (e.g., "12:30:55")

The predefined identifiers can be used in programs just like program constants. For example,

```
#define Assert(p) \
        if (!(p))   cout << __FILE__ << ": assertion on line " \
                         << __LINE__ << " failed.\n"
```

defines an assert macro for testing program invariants. Assuming that the sample call

```
Assert(ptr != 0);
```

appear in file `prog.cpp` on line 50, when the stated condition fails, the following message is displayed:

```
prog.cpp: assertion on line 50 failed.
```

□

Exercises

12.1 Define plain macros for the following:

- An infinite loop structure called `forever`.
- Pascal style `begin` and `end` keywords.
- Pascal style `if-then-else` statements.
- Pascal style `repeat-until` loop.

12.2 Define parameterized macros for the following:

- Swapping two values.
- Finding the absolute value of a number.
- Finding the center of a rectangle whose top-left and bottom-right coordinates are given (requires two macros).

Redefine the above as inline functions or function templates as appropriate.

12.3 Write directives for the following:

- Defining `Small` as an `unsigned char` when the symbol `PC` is defined, and as `unsigned short` otherwise.
- Including the file `basics.h` in another file when the symbol `CPP` is not defined.
- Including the file `debug.h` in another file when `release` is 0, or `beta.h` when `release` is 1, or `final.h` when `release` is greater than 1.

12.4 Write a macro named `When` which returns the current date and time as a string (e.g., `"25 Dec 1995, 12:30:55"`). Similarly, write a macro named `Where` which returns the current location in a file as a string (e.g., `"file.h: line 25"`).

□

Solutions to Exercises

1.1

```
#include <iostream.h>

int main (void)
{
    double fahrenheit;
    double celsius;

    cout << "Temperature in Fahrenhait: ";
    cin  >> fahrenheit;

    celsius = 5 * (fahrenheit - 32) / 9;

    cout << fahrenheit << " degrees Fahrenheit = "
         << celsius << " degrees Celsius\n";
    return 0;
}
```

1.2

```
int n = -100;                       // valid
unsigned int i = -100;              // valid
signed int = 2.9;                   // invalid: no variable name
long m = 2, p = 4;                  // valid
int 2k;                             // invalid: 2k not an identifier
double x = 2 * m;                   // valid
float y = y * 2;                        // valid (but dangerous!)
unsigned double z = 0.0;            // invalid: can't be unsigned
double d = 0.67F;                   // valid
float f = 0.52L;                        // valid
signed char = -1786;                    // invalid: no variable name
char c = '$' + 2;                   // valid
sign char h = '\111';               // invalid: 'sign' not recognized
char *name = "Peter Pan";           // valid
unsigned char *num = "276811";      // valid
```

1.3

```
identifier                          // valid
seven_11                                // valid
_unique_                                // valid
gross-income                            // invalid: - not allowed in id
gross$income                            // invalid: $ not allowed in id
2by2                                // invalid: can't start with digit
default                             // invalid: default is a keyword
average_weight_of_a_large_pizza     // valid
variable                                // valid
object.oriented                     // invalid: . not allowed in id
```

1.4

```
int     age;                        // age of a person
double  employeeIncome;             // employee income
long    wordsInDictn;               // number of words in dictionary
char    letter;                     // letter of alphabet
```



```
char    *greeting;                  // greeting message
```

2.1
```
// test if n is even:
    n%2 == 0
// test if c is a digit:
    c >= '0' && c <= '9'
// test if c is a letter:
    c >= 'a' && c <= 'z' || c >= 'A' && c <= 'Z'
// test if n is odd and positive or n is even and negative:
    n%2 != 0 && n >= 0 || n%2 == 0 && n < 0
// set the n-th bit of a long integer f to 1:
    f |= (1L << n)
// reset the n-th bit of a long integer f to 0:
    f &= ~(1L << n)
// give the absolute value of a number n:
    (n >= 0 ? n : -n)
// give the number of characters in a null-terminated string literal s:
    sizeof(s) - 1
```

2.2
```
(((n <= (p + q)) && (n >= (p - q))) || (n == 0))
(((++n) * (q--)) / ((++p) - q))
(n | ((p & q) ^ (p << (2 + q))))
((p < q) ? ((n < p) ? ((q * n) - 2) : ((q / n) + 1)) : (q - n))
```

2.3
```
double  d = 2 * int(3.14);          // initializes d to 6
long    k = 3.14 - 3;               // initializes k to 0
char    c = 'a' + 2;                // initializes c to 'c'
char    c = 'p' + 'A' - 'a';        // initializes c to 'P'
```

2.4
```
#include <iostream.h>

int main (void)
{
    long n;

    cout << "What is the value of n? ";
    cin  >> n;
    cout << "2 to the power of " << n << " = " << (1L << n) << '\n';
    return 0;
}
```

2.5
```
#include <iostream.h>

int main (void)
{
    double n1, n2, n3;

    cout << "Input three numbers: ";
    cin  >> n1 >> n2 >> n3;
    cout << (n1 <= n2 && n2 <= n3 ? "Sorted" : "Not sorted") << '\n';
    return 0;
}
```

3.1
```
#include <iostream.h>

int main (void)
{
    double height, weight;

    cout << "Person's height (in centimeters): ";
    cin >> height;
    cout << "Person's weight (in kilograms: ";
```

```
    cin >> weight;

    if (weight < height/2.5)
        cout << "Underweight\n";
    else if (height/2.5 <= weight && weight <= height/2.3)
        cout << "Normal\n";
    else
        cout << "Overweight\n";
    return 0;
}
```

3.2 It will output the message `n is negative`.

This is because the else clause is associated with the if clause immediately preceding it. The indentation in the code fragment

```
if (n >= 0)
    if (n < 10)
        cout << "n is small\n";
else
    cout << "n is negative\n";
```

is therefore misleading, because it is understood by the compiler as:

```
if (n >= 0)
    if (n < 10)
        cout << "n is small\n";
    else
        cout << "n is negative\n";
```

The problem is fixed by placing the second if within a compound statement:

```
if (n >= 0) {
    if (n < 10)
        cout << "n is small\n";
} else
    cout << "n is negative\n";
```

3.3

```
#include <iostream.h>

int main (void)
{
    int day, month, year;
    char ch;

    cout << "Input a date as dd/mm/yy: ";
    cin >> day >> ch >> month >> ch >> year;

    switch (month) {
        case 1:     cout << "January";  break;
        case 2:     cout << "February"; break;
        case 3:     cout << "March";    break;
        case 4:     cout << "April";    break;
        case 5:     cout << "May";      break;
        case 6:     cout << "June";     break;
        case 7:     cout << "July";     break;
        case 8:     cout << "August";   break;
        case 9:     cout << "September";break;
        case 10:    cout << "October";  break;
```

```
            case 11:    cout << "November"; break;
            case 12:    cout << "December"; break;
        }
        cout << ' ' << day << ", " << 1900 + year << '\n';
        return 0;
    }
```

3.4

```
    #include <iostream.h>

    int main (void)
    {
        int n;
        int factorial = 1;

        cout << "Input a positive integer: ";
        cin >> n;

        if (n >= 0) {
            for (register int i = 1; i <= n; ++i)
                factorial *= i;
            cout << "Factorial of " << n << " = " << factorial << '\n';
        }
        return 0;
    }
```

3.5

```
    #include <iostream.h>

    int main (void)
    {
        int octal, digit;
        int decimal = 0;
        int power = 1;

        cout << "Input an octal number: ";
        cin >> octal;

        for (int n = octal; n > 0; n /= 10) {    // process each digit
            digit = n % 10;                      // right-most digit
            decimal = decimal + power * digit;
            power *= 8;
        }
        cout << "Octal(" << octal << ") = Decimal(" << decimal << ")\n";
        return 0;
    }
```

3.6

```
    #include <iostream.h>

    int main (void)
    {
        for (register i = 1; i <= 9; ++i)
            for (register j = 1; j <= 9; ++j)
                cout << i << " x " << j << " = " << i*j << '\n';

        return 0;
    }
```

4.1a

```
    #include <iostream.h>

    double FahrenToCelsius (double fahren)
    {
        return 5 * (fahren - 32) / 9;
```

```
}

int main (void)
{
    double fahrenheit;

    cout << "Temperature in Fahrenhait: ";
    cin  >> fahrenheit;

    cout << fahrenheit << " degrees Fahrenheit = "
         << FahrenToCelsius(fahrenheit) << " degrees Celsius\n";
    return 0;
}
```

4.1b

```
#include <iostream.h>

char* CheckWeight (double height, double weight)
{
    if (weight < height/2.5)
        return "Underweight";
    if (height/2.5 <= weight && weight <= height/2.3)
        return "Normal";
    return "Overweight";
}

int main (void)
{
    double height, weight;

    cout << "Person's height (in centimeters): ";
    cin >> height;
    cout << "Person's weight (in kilograms: ";
    cin >> weight;
    cout << CheckWeight(height, weight) << '\n';

    return 0;
}
```

4.2 The value of `x` and `y` will be unchanged because `Swap` uses value parameters. Consequently, it swaps a copy of `x` and `y` and not the originals.

4.3 The program will output:

```
Parameter
Local
Global
Parameter
```

4.4

```
enum Bool {false, true};

void Primes (unsigned int n)
{
    Bool isPrime;

    for (register num = 2; num <= n; ++num) {
        isPrime = true;
        for (register i = 2; i < num/2; ++i)
            if (num%i == 0) {
                isPrime = false;
```

```
                    break;
                }
            if (isPrime)
                cout << num << '\n';

        }
    }
```

4.5

```
    enum Month {
        Jan, Feb, Mar, Apr, May, Jun,
        Jul, Aug, Sep, Oct, Nov, Dec
    };

    char* MonthStr (Month month)
    {
        switch (month) {
            case Jan:   return "January";
            case Feb:   return "february";
            case Mar:   return "March";
            case Apr:   return "April";
            case May:   return "May";
            case Jun:   return "June";
            case Jul:   return "July";
            case Aug:   return "August";
            case Sep:   return "September";
            case Oct:   return "October";
            case Nov:   return "November";
            case Dec:   return "December";
            default:    return "";
        }

    }
```

4.6

```
    inline int IsAlpha (char ch)
    {
        return ch >= 'a' && ch <= 'z' || ch >= 'A' && ch <= 'Z';
    }
```

4.7

```
    int Power (int base, unsigned int exponent)
    {
        return (exponent <= 0)
            ? 1
            : base * Power(base, exponent - 1);
    }
```

4.8

```
    double Sum (int n, double val ...)
    {
        va_list args;               // argument list
        double  sum = 0;

        va_start(args, val);        // initialize args

        while (n-- > 0) {
            sum += val;
            val = va_arg(args, double);
        }
        va_end(args);               // clean up args
        return sum;

    }
```

5.1

```
void ReadArray (double nums[], const int size)
{
    for (register i = 0; i < size; ++i) {
        cout << "nums[" << i << "] = ";
        cin >> nums[i];
    }
}

void WriteArray (double nums[], const int size)
{
    for (register i = 0; i < size; ++i)
        cout << nums[i] << '\n';
}
```

5.2

```
void Reverse (double nums[], const int size)
{
    double temp;

    for (register i = 0; i < size/2; ++i) {
        temp = nums[i];
        nums[i] = nums[size - i - 1];
        nums[size - i - 1] = temp;
    }
}
```

5.3

```
double contents[][4] = {
    { 12, 25, 16, 0.4 },
    { 22,  4,  8, 0.3 },
    { 28,  5,  9, 0.5 },
    { 32,  7,  2, 0.2 }
};

void WriteContents (const double *contents,
                    const int rows, const int cols)
{
    for (register i = 0; i < rows; ++i) {
        for (register j = 0; j < cols; ++j)
            cout << *(contents + i * rows + j) << ' ';
        cout << '\n';
    }
}
```

5.4

```
enum Bool {false, true};

void ReadNames (char *names[], const int size)
{
    char name[128];

    for (register i = 0; i < size; ++i) {
        cout << "names[" << i << "] = ";
        cin >> name;
        names[i] = new char[strlen(name) + 1];
        strcpy(names[i], name);
    }
}

void WriteNames (char *names[], const int size)
{
    for (register i = 0; i < size; ++i)
        cout << names[i] << '\n';
}
```



```
void BubbleSort (char *names[], const int size)
{
    Bool swapped;
    char *temp;

    do {
        swapped = false;
        for (register i = 0; i < size - 1; ++i) {
            if (strcmp(names[i], names[i+1]) > 0 ) {
                temp = names[i];
                names[i] = names[i+1];
                names[i+1] = temp;
                swapped = true;
            }
        }
    } while (swapped);
}
```

5.5

```
char* ReverseString (char *str)
{
    int len = strlen(str);
    char *result = new char[len + 1];
    char *res = result + len;

    *res-- = '\0';
    while (*str)
        *res-- = *str++;
    return result;
}
```

5.6

```
typedef int (*Compare)(const char*, const char*);

void BubbleSort (char *names[], const int size, Compare comp)
{
    Bool swapped;
    char *temp;

    do {
        swapped = false;
        for (register i = 0; i < size - 1; ++i) {
            if (comp(names[i], names[i+1]) > 0 ) {
                temp = names[i];
                names[i] = names[i+1];
                names[i+1] = temp;
                swapped = true;
            }
        }
    } while (swapped);
}
```

5.7

```
typedef void (*SwapFun)(double, double);
    SwapFun Swap;
typedef char *Table[];
    Table table;
typedef char *&Name;
    Name name;
typedef unsigned long *Values[10][20];
    Values values;
```

6.1 Declaring `Set` parameters as references avoids their being copied in a call. Call-by-reference is generally more efficient than call-by-value when the objects involved are larger than the built-in type objects.

6.2
```
class Complex {
public:
            Complex (double r = 0, double i = 0)
                    {real = r; imag = i;}
    Complex Add     (Complex &c);
    Complex Subtract(Complex &c);
    Complex Multiply(Complex &c);
    void    Print       (void);
private:
    double  real;   // real part
    double  imag;   // imaginary part
};

Complex Complex::Add (Complex &c)
{
    return Complex(real + c.real, imag + c.imag);
}

Complex Complex::Subtract (Complex &c)
{
    return Complex(real - c.real, imag - c.imag);
}

Complex Complex::Multiply (Complex &c)
{
    return Complex( real * c.real - imag * c.imag,
                    imag * c.real + real * c.imag);
}

void Complex::Print (void)
{
    cout << real << " + i" << imag << '\n';
}
```

6.3
```
#include <iostream.h>
#include <string.h>

const int end = -1;     // denotes the end of the list

class Menu {
public:
            Menu    (void)      {first = 0;}
            ~Menu   (void);
    void    Insert  (const char *str, const int pos = end);
    void    Delete  (const int pos = end);
    int     Choose  (void);

private:

    class Option {
    public:
                    Option  (const char*);
                    ~Option (void)  {delete name;}
        const char* Name    (void)  {return name;}
        Option*&        Next    (void)  {return next;}
    private:
        char    *name;  // option name
        Option  *next;  // next option
```



```
    };

    Option  *first;     // first option in the menu
};

Menu::Option::Option (const char* str)
{
    name = new char [strlen(str) + 1];
    strcpy(name, str);
    next = 0;
}

Menu::~Menu (void)
{
    Menu::Option *handy, *next;

    for (handy = first; handy != 0; handy = next) {
        next = handy->Next();
        delete handy;
    }
}

void Menu::Insert (const char *str, const int pos)
{
    Menu::Option *option = new Menu::Option(str);
    Menu::Option *handy, *prev = 0;
    int idx = 0;

    // set prev to point to before the insertion position:
    for (handy = first; handy != 0 && idx++ != pos; handy = handy-
>Next())
        prev = handy;

    if (prev == 0) {            // empty list
       option->Next() = first;  // first entry
       first = option;
    } else {                    // insert
        option->Next() = handy;
        prev->Next() = option;
    }
}

void Menu::Delete (const int pos)
{
    Menu::Option *handy, *prev = 0;
    int idx = 0;

    // set prev to point to before the deletion position:
    for (handy = first;
        handy != 0 && handy->Next() != 0 && idx++ != pos;
        handy = handy->Next())
        prev = handy;

    if (handy != 0) {
        if (prev == 0)              // it's the first entry
            first = handy->Next();
        else                        // it's not the first
            prev->Next() = handy->Next();
        delete handy;
    }
}

int Menu::Choose (void)
{
    int n, choice;
```

```
    Menu::Option *handy = first;

    do {
        n = 0;
        for (handy = first; handy != 0; handy = handy->Next())
            cout << ++n << ". " << handy->Name() << '\n';
        cout << "Option? ";
        cin >> choice;
    } while (choice <= 0 || choice > n);

    return choice;
}
```

6.4

```
#include <iostream.h>

const int   maxCard = 10;
enum  Bool  {false, true};


class Set {
public:
            Set         (void)      { first = 0; }
            ~Set            (void);
    int     Card        (void);
    Bool    Member      (const int) const;
    void    AddElem     (const int);
    void    RmvElem     (const int);
    void    Copy            (Set&);
    Bool    Equal       (Set&);
    void    Intersect   (Set&, Set&);
    void    Union       (Set&, Set&);
    void    Print       (void);

private:

    class Element {
    public:
                    Element (const int val) {value = val; next = 0;}
        int         Value   (void)          {return value;}
        Element*&   Next    (void)          {return next;}
    private:
        int         value;  // element value
        Element     *next;  // next element
    };

    Element *first;         // first element in the list
};

Set::~Set (void)
{
    Set::Element *handy, *next;

    for (handy = first; handy != 0; handy = next) {
        next = handy->Next();
        delete handy;
    }
}

int Set::Card (void)
{
    Set::Element *handy;
    int card = 0;

    for (handy = first; handy != 0; handy = handy->Next())
```



```
        ++card;
    return card;
}

Bool Set::Member (const int elem) const
{
    Set::Element *handy;

    for (handy = first; handy != 0; handy = handy->Next())
        if (handy->Value() == elem)
            return true;
    return false;
}

void Set::AddElem (const int elem)
{
    if (!Member(elem)) {
        Set::Element *option = new Set::Element(elem);
        option->Next() = first;      // prepend
        first = option;
    }
}

void Set::RmvElem (const int elem)
{
    Set::Element *handy, *prev = 0;
    int idx = 0;

    // set prev to point to before the deletion position:
    for (handy = first;
        handy != 0 && handy->Next() != 0 && handy->Value() != elem;
        handy = handy->Next())
        prev = handy;

    if (handy != 0) {
        if (prev == 0)               // it's the first entry
            first = handy->Next();
        else                         // it's not the first
            prev->Next() = handy->Next();
        delete handy;
    }
}

void Set::Copy (Set &set)
{
    Set::Element *handy;

    for (handy = first; handy != 0; handy = handy->Next())
        set.AddElem(handy->Value());
}

Bool Set::Equal (Set &set)
{
    Set::Element *handy;

    if (Card() != set.Card())
        return false;
    for (handy = first; handy != 0; handy = handy->Next())
        if (!set.Member(handy->Value()))
            return false;
    return true;
}

void Set::Intersect (Set &set, Set &res)
{
```

```
    Set::Element *handy;

    for (handy = first; handy != 0; handy = handy->Next())
        if (set.Member(handy->Value()))
            res.AddElem(handy->Value());
}

void Set::Union (Set &set, Set &res)
{
    Copy(res);
    set.Copy(res);
}

void Set::Print (void)
{
    Set::Element *handy;

    cout << '{';
    for (handy = first; handy != 0; handy = handy->Next()) {
        cout << handy->Value();
        if (handy->Next() != 0)
            cout << ',';
    }
    cout << "}\n";
}
```

6.5

```
#include <iostream.h>
#include <string.h>

enum Bool {false, true};
typedef char *String;

class BinNode;
class BinTree;

class Sequence {
public:
                Sequence    (const int size);
                ~Sequence   (void)      {delete entries;}
    void        Insert      (const char*);
    void        Delete      (const char*);
    Bool        Find        (const char*);
    void        Print       (void);
    int         Size        (void)  {return used;}
friend  BinNode* BinTree::MakeTree (Sequence &seq, int low, int high);

protected:
    char        **entries;      // sorted array of string entries
    const int   slots;          // number of sequence slots
    int         used;           // number of slots used so far

};

void
Sequence::Insert (const char *str)
{
    if (used >= slots)
        return;
    for (register i = 0; i < used; ++i) {
        if (strcmp(str,entries[i]) < 0)
            break;
    }
    for (register j = used; j > i; --j)
        entries[j] = entries[j-1];
```



```
    entries[i] = new char[strlen(str) + 1];
    strcpy(entries[i], str);
    ++used;
}

void
Sequence::Delete (const char *str)
{
    for (register i = 0; i < used; ++i) {
        if (strcmp(str,entries[i]) == 0) {
            delete entries[i];
            for (register j = i; j < used-1; ++j)
                entries[j] = entries[j+1];
            --used;
            break;
        }
    }
}

Bool
Sequence::Find (const char *str)
{
    for (register i = 0; i < used; ++i)
        if (strcmp(str,entries[i]) == 0)
            return true;
    return false;
}

void
Sequence::Print (void)
{
    cout << '[';
    for (register i = 0; i < used; ++i) {
        cout << entries[i];
        if (i < used-1)
            cout << ',';
    }
    cout << "]\n";
}
```

6.6

```
#include <iostream.h>
#include <string.h>

enum Bool {false,true};

class BinNode {
public:
                BinNode     (const char*);
                ~BinNode    (void)  {delete value;}
    char*&      Value       (void)  {return value;}
    BinNode*&   Left        (void)  {return left;}
    BinNode*&   Right       (void)  {return right;}

    void        FreeSubtree (BinNode *subtree);
    void        InsertNode  (BinNode *node, BinNode *&subtree);
    void        DeleteNode  (const char*, BinNode *&subtree);
const BinNode*  FindNode    (const char*, const BinNode *subtree);
    void        PrintNode   (const BinNode *node);

private:
    char    *value;     // node value
    BinNode *left;      // pointer to left child
    BinNode *right;     // pointer to right child
};
```

```
class BinTree {
public:
            BinTree (void)              {root = 0;}
            BinTree (Sequence &seq);
            ~BinTree(void)              {root->FreeSubtree(root);}
    void    Insert  (const char *str);
    void    Delete  (const char *str)   {root->DeleteNode(str, root);}
    Bool    Find    (const char *str)   {return root->FindNode(str,
root) != 0;}
    void    Print   (void)      {root->PrintNode(root);  cout << '\n';}

protected:
    BinNode* root;          // root node of the tree
};

BinNode::BinNode (const char *str)
{
    value = new char[strlen(str) + 1];
    strcpy(value, str);
    left = right = 0;
}

void
BinNode::FreeSubtree (BinNode *node)
{
    if (node != 0) {
        FreeSubtree(node->left);
        FreeSubtree(node->right);
        delete node;
    }
}

void
BinNode::InsertNode (BinNode *node, BinNode *&subtree)
{
    if (subtree == 0)
        subtree = node;
    else if (strcmp(node->value, subtree->value) <= 0)
        InsertNode(node, subtree->left);
    else
        InsertNode(node, subtree->right);
}

void
BinNode::DeleteNode (const char *str, BinNode *&subtree)
{
    int cmp;

    if (subtree == 0)
        return;
    if ((cmp = strcmp(str, subtree->value)) < 0)
        DeleteNode(str, subtree->left);
    else if (cmp > 0)
        DeleteNode(str, subtree->right);
    else {
        BinNode* handy = subtree;
        if (subtree->left == 0)     // no left subtree
            subtree = subtree->right;
        else if (subtree->right == 0)   // no right subtree
            subtree = subtree->left;
        else {                          // left and right subtree
            subtree = subtree->right;
            // insert left subtree into right subtree:
            InsertNode(subtree->left, subtree->right);
```



```
        }
        delete handy;
    }
}

const BinNode*
BinNode::FindNode (const char *str, const BinNode *subtree)
{
    int cmp;

    return (subtree == 0)
            ? 0
            : ((cmp = strcmp(str, subtree->value)) < 0
                ? FindNode(str, subtree->left)
                : (cmp > 0
                    ? FindNode(str, subtree->right)
                    : subtree));
}

void
BinNode::PrintNode (const BinNode *node)
{
    if (node != 0) {
        PrintNode(node->left);
        cout << node->value << ' ';
        PrintNode(node->right);
    }
}

BinTree::BinTree (Sequence &seq)
{
    root = MakeTree(seq, 0, seq.Size() - 1);
}

void
BinTree::Insert (const char *str)
{
    root->InsertNode(new BinNode(str), root);
}
```

6.7

```
class Sequence {
    //...
friend  BinNode* BinTree::MakeTree (Sequence &seq, int low, int high);
};

class BinTree {
public:
    //...
            BinTree (Sequence &seq);
    //...
    BinNode*    MakeTree (Sequence &seq, int low, int high);
};

BinTree::BinTree (Sequence &seq)
{
    root = MakeTree(seq, 0, seq.Size() - 1);
}

BinNode*
BinTree::MakeTree (Sequence &seq, int low, int high)
{
    int mid = (low + high) / 2;
    BinNode* node = new BinNode(seq.entries[mid]);
```

```
    node->Left() = (mid == low ? 0 : MakeTree(seq, low, mid-1));
    node->Right() = (mid == high ? 0 : MakeTree(seq, mid+1, high));
    return node;
}
```

6.8 A static data member is used to keep track of the last allocated ID (see `lastId` below).

```
class Menu {
public:
    //...
    int     ID      (void)      {return id;}
private:
    //...
    int     id;                 // menu ID
    static int  lastId;         // last allocated ID
};

int Menu::lastId = 0;
```

6.9

```
#include <iostream.h>
#include <string.h>

const int end = -1;     // denotes the end of the list
class Option;

class Menu {
public:
            Menu    (void)      {first = 0; id = lastId++;}
            ~Menu   (void);
    void    Insert  (const char *str, const Menu *submenu, const int
pos = end);
    void    Delete  (const int pos = end);
    int     Print   (void);
    int     Choose  (void)  const;
    int     ID      (void)      {return id;}

private:

    class Option {
    public:
                    Option  (const char*, const Menu* = 0);
                    ~Option (void);
        const char* Name    (void)  {return name;}
        const Menu* Submenu (void)  {return submenu;}
        Option*&    Next    (void)  {return next;}
        void        Print   (void);
        int         Choose  (void)  const;
    private:
        char        *name;      // option name
        const Menu  *submenu;   // submenu
        Option      *next;      // next option
    };

    Option  *first;             // first option in the menu
    int     id;                 // menu ID

    static int  lastId;         // last allocated ID
};

Menu::Option::Option (const char *str, const Menu *menu) :
submenu(menu)
{
```



```
    name = new char [strlen(str) + 1];
    strcpy(name, str);
    next = 0;
}

Menu::Option::~Option (void)
{
    delete name;
    delete submenu;
}

void Menu::Option::Print (void)
{
    cout << name;
    if (submenu != 0)
        cout << " ->";
    cout << '\n';
}

int Menu::Option::Choose (void) const
{
    if (submenu == 0)
        return 0;
    else
        return submenu->Choose();
}

int Menu::lastId = 0;

Menu::~Menu (void)
{
    Menu::Option *handy, *next;

    for (handy = first; handy != 0; handy = next) {
        next = handy->Next();
        delete handy;
    }
}

void Menu::Insert (const char *str, const Menu *submenu, const int pos)
{
    Menu::Option *option = new Option(str, submenu);
    Menu::Option *handy, *prev = 0;
    int idx = 0;

    // set prev to point to before the insertion position:
    for (handy = first; handy != 0 && idx++ != pos; handy = handy-
>Next())
        prev = handy;

    if (prev == 0) {            // empty list
       option->Next() = first;  // first entry
       first = option;
    } else {                    // insert
        option->Next() = handy;
        prev->Next() = option;
    }
}

void Menu::Delete (const int pos)
{
    Menu::Option *handy, *prev = 0;
    int idx = 0;

    // set prev to point to before the deletion position:
```

```
    for (handy = first;
        handy != 0 && handy->Next() != 0 && idx++ != pos;
        handy = handy->Next())
        prev = handy;

    if (handy != 0) {
        if (prev == 0)              // it's the first entry
            first = handy->Next();
        else                        // it's not the first
            prev->Next() = handy->Next();
        delete handy;
    }
}

int Menu::Print (void)
{
    int n = 0;
    Menu::Option *handy = first;

    for (handy = first; handy != 0; handy = handy->Next()) {
        cout << ++n << ". ";
        handy->Print();
    }
    return n;
}

int Menu::Choose (void) const
{
    int choice, n;

    do {
        n = Print();
        cout << "Option? ";
        cin >> choice;
    } while (choice <= 0 || choice > n);

    Menu::Option *handy = first;
    n = 1;

    // move to the chosen option:
    for (handy = first; n != choice && handy != 0; handy = handy-
>Next())
        ++n;
    // choose the option:
    n = handy->Choose();

    return (n == 0 ? choice : n);

}
```

7.1

```
#include <string.h>

const int Max (const int x, const int y)
{
    return x >= y ? x : y;
}

const double Max (const double x, const double y)
{
    return x >= y ? x : y;
}

const char* Max (const char *x, const char *y)
{
```



```
    return strcmp(x,y) >= 0 ? x : y;
}
```

7.2

```
class Set {
    //...
    friend Set  operator -  (Set&, Set&);       // difference
    friend Bool operator <= (Set&, Set&);       // subset
    //...
};

Set operator - (Set &set1, Set &set2)
{
    Set res;

    for (register i = 0; i < set1.card; ++i)
        if (!(set1.elems[i] & set2))
            res.elems[res.card++] = set1.elems[i];
    return res;
}

Bool operator <= (Set &set1, Set &set2)
{
    if (set1.card > set2.card)
        return false;
    for (register i = 0; i < set1.card; ++i)
        if (!(set1.elems[i] & set2))
            return false;
    return true;
}
```

7.3

```
class Binary {
//...
friend  Binary  operator -   (const Binary, const Binary);
        int     operator []  (const int n)
                {return bits[15-n] == '1' ? 1 : 0;}
//...
};

Binary operator - (const Binary n1, const Binary n2)
{
    unsigned borrow = 0;
    unsigned value;
    Binary res = "0";

    for (register i = 15; i >= 0; --i) {
        value = (n1.bits[i] == '0' ? 0 : 1) -
                (n2.bits[i] == '0' ? 0 : 1) + borrow;
        res.bits[i] = (value == -1 || value == 1 ? '1': '0');
        borrow = (value == -1 || borrow != 0 && value == 1 ? 1 : 0);
    }
    return res;
}
```

7.4

```
#include <iostream.h>

class Matrix {
public:
                Matrix      (const int rows, const int cols);
                Matrix      (const Matrix&);
                ~Matrix     (void);
        double& operator () (const int row, const int col);
        Matrix& operator =  (const Matrix&);
```

```
friend ostream& operator << (ostream&, Matrix&);
friend  Matrix  operator +  (Matrix&, Matrix&);
friend  Matrix  operator -  (Matrix&, Matrix&);
friend  Matrix  operator *  (Matrix&, Matrix&);
        int     Rows        (void)  {return rows;}
        int     Cols        (void)  {return cols;}
protected:
        class Element {         // nonzero element
        public:
                        Element (const int row, const int col, double);
            const int   Row     (void)  {return row;}
            const int   Col     (void)  {return col;}
            double&     Value   (void)  {return value;}
            Element*&   Next    (void)  {return next;}
            Element*    CopyList(Element *list);
            void        DeleteList  (Element *list);
        private:
            const int   row, col;   // row and column of element
            double      value;      // element value
            Element     *next;      // pointer to next element
        };

        double& InsertElem  (Element *elem, const int row, const int
col);

        int     rows, cols; // matrix dimensions
        Element *elems;     // linked-list of elements
};

Matrix::Element::Element (const int r, const int c, double val)
           : row(r), col(c)
{
    value = val;
    next = 0;
}

Matrix::Element* Matrix::Element::CopyList (Element *list)
{
    Element *prev = 0;
    Element *first = 0;
    Element *copy;

    for (; list != 0; list = list->Next()) {
        copy = new Element(list->Row(), list->Col(), list->Value());
        if (prev == 0)
            first = copy;
        else
            prev->Next() = copy;
        prev = copy;
    }
    return first;
}

void Matrix::Element::DeleteList (Element *list)
{
    Element *next;

    for (; list != 0; list = next) {
        next = list->Next();
        delete list;
    }
}

// InsertElem creates a new element and inserts it before
// or after the element denoted by elem.
```



```
double& Matrix::InsertElem (Element *elem, const int row, const int
col)
{
    Element* newElem = new Element(row, col, 0.0);

    if (elem == elems && (elems == 0 || row < elems->Row() ||
                          row == elems->Row() && col < elems->Col())) {
        // insert in front of the list:
        newElem->Next() = elems;
        elems = newElem;
    } else {
        // insert after elem:
        newElem->Next() = elem->Next();
        elem->Next() = newElem;
    }
    return newElem->Value();
}

Matrix::Matrix (const int rows, const int cols)
{
    Matrix::rows = rows;
    Matrix::cols = cols;
    elems = 0;
}

Matrix::Matrix (const Matrix &m)
{
    rows = m.rows;
    cols = m.cols;
    elems = m.elems->CopyList(m.elems);
}

Matrix::~Matrix (void)
{
    elems->DeleteList(elems);
}

Matrix& Matrix::operator = (const Matrix &m)
{
    elems->DeleteList(elems);
    rows = m.rows;
    cols = m.cols;
    elems = m.elems->CopyList(m.elems);
    return *this;
}

double& Matrix::operator () (const int row, const int col)
{
    if (elems == 0 || row < elems->Row() ||
        row == elems->Row() && col < elems->Col())
        // create an element and insert in front:
        return InsertElem(elems, row, col);

    // check if it's the first element in the list:
    if (row == elems->Row() && col == elems->Col())
        return elems->Value();

    // search the rest of the list:
    for (Element *elem = elems; elem->Next() != 0; elem = elem->Next())
        if (row == elem->Next()->Row()) {
            if (col == elem->Next()->Col())
                return elem->Next()->Value();   // found it!
            else if (col < elem->Next()->Col())
                break;                          // doesn't exist
```

```
        } else if (row < elem->Next()->Row())
            break;                              // doesn't exist
    // create new element and insert just after elem:
    return InsertElem(elem, row, col);
}

ostream& operator << (ostream &os, Matrix &m)
{
    Matrix::Element *elem = m.elems;

    for (register row = 1; row <= m.rows; ++row) {
        for (register col = 1; col <= m.cols; ++col)
            if (elem != 0 && elem->Row() == row && elem->Col() == col)
{
                os << elem->Value() << '\t';
                elem = elem->Next();
            } else
                os << 0.0 << '\t';
        os << '\n';
    }
    return os;
}

Matrix operator + (Matrix &p, Matrix &q)
{
    Matrix m(p.rows, q.cols);

    // copy p:
    for (Matrix::Element *pe = p.elems; pe != 0; pe = pe->Next())
        m(pe->Row(), pe->Col()) = pe->Value();
    // add q:
    for (Matrix::Element *qe = q.elems; qe != 0; qe = qe->Next())
        m(qe->Row(), qe->Col()) += qe->Value();
    return m;
}

Matrix operator - (Matrix &p, Matrix &q)
{
    Matrix m(p.rows, q.cols);

    // copy p:
    for (Element *pe = p.elems; pe != 0; pe = pe->Next())
        m(pe->Row(), pe->Col()) = pe->Value();
    // subtract q:
    for (Element *qe = q.elems; qe != 0; qe = qe->Next())
        m(qe->Row(), qe->Col()) -= qe->Value();
    return m;
}

Matrix operator * (Matrix &p, Matrix &q)
{
    Matrix m(p.rows, q.cols);

    for (Element *pe = p.elems; pe != 0; pe = pe->Next())
        for (Element *qe = q.elems; qe != 0; qe = qe->Next())
           if (pe->Col() == qe->Row())
              m(pe->Row(),qe->Col()) += pe->Value() * qe->Value();
    return m;
}
```

7.5

```
#include <string.h>
#include <iostream.h>

class String {
```



```
public:
                    String      (const char*);
                    String      (const String&);
                    String      (const short);
                    ~String     (void);
        String&     operator =  (const char*);
        String&     operator =  (const String&);
        char&       operator [] (const short);
        int         Length      (void)      {return(len);}
friend  String      operator +  (const String&, const String&);
friend  ostream&    operator << (ostream&, String&);

protected:
    char        *chars;     // string characters
    short       len;        // length of chars
};

String::String (const char *str)
{
    len = strlen(str);
    chars = new char[len + 1];
    strcpy(chars, str);
}

String::String (const String &str)
{
    len = str.len;
    chars = new char[len + 1];
    strcpy(chars, str.chars);
}

String::String (const short size)
{
    len = size;
    chars = new char[len + 1];
    chars[0] = '\0';
}

String::~String (void)
{
    delete chars;
}

String& String::operator = (const char *str)
{
    short   strLen = strlen(str);
    if (len != strLen) {
        delete chars;
        len = strLen;
        chars = new char[strLen + 1];
    }
    strcpy(chars, str);
    return(*this);
}

String& String::operator = (const String &str)
{
    if (this != &str) {
        if (len != str.len) {
            delete chars;
            len = str.len;
            chars = new char[str.len + 1];
        }
        strcpy(chars, str.chars);
    }
```

```
    return(*this);
}

char& String::operator [] (const short index)
{
    static char dummy = '\0';
    return(index >= 0 && index < len ? chars[index] : dummy);
}

String operator + (const String &str1, const String &str2)
{
    String result(str1.len + str2.len);

    strcpy(result.chars, str1.chars);
    strcpy(result.chars + str1.len, str2.chars);
    return(result);
}

ostream& operator <<    (ostream &out, String &str)
{
    out << str.chars;
    return(out);
}
```

7.6

```
#include <string.h>
#include <iostream.h>

enum Bool {false, true};
typedef unsigned char uchar;

class BitVec {
public:
             BitVec         (const short dim);
             BitVec         (const char* bits);
             BitVec         (const BitVec&);
             ~BitVec        (void)  { delete vec; }
    BitVec& operator =      (const BitVec&);
    BitVec& operator &=     (const BitVec&);
    BitVec& operator |=     (const BitVec&);
    BitVec& operator ^=     (const BitVec&);
    BitVec& operator <<=(const short);
    BitVec& operator >>=(const short);
    int      operator []    (const short idx);
    void     Set            (const short idx);
    void     Reset          (const short idx);

    BitVec  operator ~      ();
    BitVec  operator &      (const BitVec&);
    BitVec  operator |      (const BitVec&);
    BitVec  operator ^      (const BitVec&);
    BitVec  operator <<     (const short n);
    BitVec  operator >>     (const short n);
    Bool     operator ==    (const BitVec&);
    Bool     operator !=    (const BitVec&);

friend  ostream& operator << (ostream&, BitVec&);

protected:
    uchar   *vec;           // vector of 8*bytes bits
    short    bytes;         // bytes in the vector

};

// set the bit denoted by idx to 1
```



```
inline void BitVec::Set (const short idx)
{
    vec[idx/8] |= (1 << idx%8);
}

// reset the bit denoted by idx to 0
inline void BitVec::Reset (const short idx)
{
    vec[idx/8] &= ~(1 << idx%8);
}

inline BitVec& BitVec::operator &= (const BitVec &v)
{
    return (*this) = (*this) & v;
}

inline BitVec& BitVec::operator |= (const BitVec &v)
{
    return (*this) = (*this) | v;
}

inline BitVec& BitVec::operator ^= (const BitVec &v)
{
    return (*this) = (*this) ^ v;
}

inline BitVec& BitVec::operator <<= (const short n)
{
    return (*this) = (*this) << n;
}

inline BitVec& BitVec::operator >>= (const short n)
{
    return (*this) = (*this) >> n;
}

// return the bit denoted by idx
inline int BitVec::operator [] (const short idx)
{
    return vec[idx/8] & (1 << idx%8) ? true : false;
}

inline Bool BitVec::operator != (const BitVec &v)
{
    return *this == v ? false : true;
}

BitVec::BitVec (const short dim)
{
    bytes = dim / 8 + (dim % 8 == 0 ? 0 : 1);
    vec = new uchar[bytes];

    for (register i = 0; i < bytes; ++i)
        vec[i] = 0;                 // all bits are initially zero
}

BitVec::BitVec (const char *bits)
{
    int len = strlen(bits);
    bytes = len / 8 + (len % 8 == 0 ? 0 : 1);
    vec = new uchar[bytes];

    for (register i = 0; i < bytes; ++i)
        vec[i] = 0;                 // initialize all bits to zero
```

```
    for (i = len - 1; i >= 0; --i)
        if (*bits++ == '1')         // set the 1 bits
            vec[i/8] |= (1 << (i%8));
}

BitVec::BitVec (const BitVec &v)
{
    bytes = v.bytes;
    vec = new uchar[bytes];
    for (register i = 0; i < bytes; ++i) // copy bytes
        vec[i] = v.vec[i];
}

BitVec& BitVec::operator = (const BitVec& v)
{
    for (register i = 0; i < (v.bytes < bytes ? v.bytes : bytes); ++i)
        vec[i] = v.vec[i];          // copy bytes
    for (; i < bytes; ++i)          // extra bytes in *this
        vec[i] = 0;
    return *this;
}

// bitwise COMPLEMENT
BitVec BitVec::operator ~ (void)
{
    BitVec r(bytes * 8);
    for (register i = 0; i < bytes; ++i)
        r.vec[i] = ~vec[i];
    return r;
}

// bitwise AND
BitVec BitVec::operator & (const BitVec &v)
{
    BitVec r((bytes > v.bytes ? bytes : v.bytes) * 8);
    for (register i = 0; i < (bytes < v.bytes ? bytes : v.bytes); ++i)
        r.vec[i] = vec[i] & v.vec[i];
    return r;
}

// bitwise OR
BitVec BitVec::operator | (const BitVec &v)
{
    BitVec r((bytes > v.bytes ? bytes : v.bytes) * 8);
    for (register i = 0; i < (bytes < v.bytes ? bytes : v.bytes); ++i)
        r.vec[i] = vec[i] | v.vec[i];
    return r;
}

// bitwise exclusive-OR
BitVec BitVec::operator ^ (const BitVec &v)
{
    BitVec r((bytes > v.bytes ? bytes : v.bytes) * 8);
    for (register i = 0; i < (bytes < v.bytes ? bytes : v.bytes); ++i)
        r.vec[i] = vec[i] ^ v.vec[i];
    return r;
}

// SHIFT LEFT by n bits
BitVec BitVec::operator << (const short n)
{
    BitVec r(bytes * 8);
    int     zeros = n / 8;  // bytes on the left to become zero
    int     shift = n % 8;  // left shift for remaining bytes
    register i;
```



```
    for (i = bytes - 1; i >= zeros; --i)// shift bytes left
        r.vec[i] = vec[i - zeros];

    for (; i >= 0; --i)                     // zero left bytes
        r.vec[i] = 0;
    unsigned char prev = 0;

    for (i = zeros; i < r.bytes; ++i) {     // shift bits left
        r.vec[i] = (r.vec[i] << shift) | prev;
        prev = vec[i - zeros] >> (8 - shift);
    }
    return r;
}

// SHIFT RIGHT by n bits
BitVec BitVec::operator >> (const short n)
{
    BitVec r(bytes * 8);
    int     zeros = n / 8;          // bytes on the right to become
zero
    int     shift = n % 8;          // right shift for remaining bytes
    register i;

    for (i = 0; i < bytes - zeros; ++i)     // shift bytes right
        r.vec[i] = vec[i + zeros];
    for (; i < bytes; ++i)                  // zero right bytes
        r.vec[i] = 0;

    uchar prev = 0;
    for (i = r.bytes - zeros - 1; i >= 0; --i) { // shift bits right
        r.vec[i] = (r.vec[i] >> shift) | prev;
        prev = vec[i + zeros] << (8 - shift);
    }
    return r;
}

Bool BitVec::operator ==  (const BitVec &v)
{
    int     smaller = bytes < v.bytes ? bytes : v.bytes;
    register i;

    for (i = 0; i < smaller; ++i)       // compare bytes
        if (vec[i] != v.vec[i])
            return false;
    for (i = smaller; i < bytes; ++i)   // extra bytes in first operand
        if (vec[i] != 0)
            return false;
    for (i = smaller; i < v.bytes; ++i) // extra bytes in second
operand
        if (v.vec[i] != 0)
            return false;
    return true;
}

ostream& operator << (ostream &os, BitVec &v)
{
    const int maxBytes = 256;
    char  buf[maxBytes * 8 + 1];
    char  *str = buf;
    int   n = v.bytes > maxBytes ? maxBytes : v.bytes;

    for (register i = n-1; i >= 0; --i)
        for (register j = 7; j >= 0; --j)
            *str++ = v.vec[i] & (1 << j) ? '1' : '0';
```

```
    *str = '\0';
    os << buf;
    return os;
}
```

8.1

```
#include "bitvec.h"

enum Month {
    Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec
};

inline Bool LeapYear(const short year)  {return year%4 == 0;}

class Year : public BitVec {
public:
            Year    (const short year);
    void    WorkDay (const short day);  // set day as work day
    void    OffDay  (const short day);  // set day as off day
    Bool    Working (const short day);  // true if a work day
    short   Day     (const short day,   // convert date to day
                     const Month month, const short year);
protected:
    short   year;   // calendar year
};

Year::Year (const short year) : BitVec(366)
{
    Year::year = year;
}

void Year::WorkDay (const short day)
{
    Set(day);
}

void Year::OffDay (const short day)
{
    Reset(day);
}

Bool Year::Working (const short day)
{
    return (*this)[day] == 1 ? true : false;
}

short Year::Day (const short day, const Month month, const short year)
{
    static short days[12] = {
        31, 28, 31, 30, 31, 30, 31, 31, 20, 31, 30, 31
    };
    days[Feb] = LeapYear(year) ? 29 : 28;

    int res = day;
    for (register i = Jan; i < month; ++i)
        res += days[i];
    return res;
}
```

8.2

```
#include <stdlib.h>
#include <time.h>
#include "matrix.h"

inline double Abs(double n) {return n >= 0 ? n : -n;}
```



```
class LinEqns : public Matrix {
public:
            LinEqns     (const int n, double *soln);
    void    Generate    (const int coef);
    void    Solve       (void);
private:
    Matrix  solution;
};

LinEqns::LinEqns (const int n, double* soln)
        : Matrix(n, n+1), solution(n, 1)
{
    for (register r = 1; r <= n; ++r)
        solution(r, 1) = soln[r - 1];
}

void LinEqns::Generate (const int coef)
{
    int mid = coef / 2;

    srand((unsigned int) time(0));  // set random seed

    for (register r = 1; r <= Rows(); ++r) {
        (*this)(r, Cols()) = 0.0;   // initialize right-hand side

        // generate equations whose coefficients
        // do not exceed coef:

        for (register c = 1; c < Cols(); ++c) {
            (*this)(r, c) = (double) (mid - random(1000) % coef);
            (*this)(r, Cols()) += (*this)(r, c) * solution(c, 1);
        }
    }
}

// solve equations using Gaussian elimination

void LinEqns::Solve (void)
{
    double const epsilon = 1e-5;// 'almost zero' quantity
    double temp;
    int diag, piv, r, c;

    for (diag = 1; diag <= Rows(); ++diag) {// diagonal
        piv = diag;                                 // pivot
        for (r = diag + 1; r <= Rows(); ++r)// upper triangle
            if (Abs((*this)(piv, diag)) < Abs((*this)(r, diag)))
                piv = r;                            // choose new pivot

        // make sure there is a unique solution:
        if (Abs((*this)(piv, diag)) < epsilon) {
            if (Abs((*this)(diag, Cols())) < epsilon)
                cout << "infinite solutions\n";
            else
                cout << "no solution\n";
            return;
        }
        if (piv != diag) {
            // swap pivit with diagonal:
            for (c = 1; c <= Cols(); ++c) {
                temp = (*this)(diag, c);
                (*this)(diag, c) = (*this)(piv, c);
                (*this)(piv, c) = temp;
            }
        }
```

```
            // normalise diag row so that m[diag, diag] = 1:
            temp = (*this)(diag, diag);
            (*this)(diag, diag) = 1.0;
            for (c = diag + 1; c <= Cols(); ++c)
                (*this)(diag, c) = (*this)(diag, c) / temp;

            // now eliminate entries below the pivot:
            for (r = diag + 1; r <= Rows(); ++r) {
                double factor = (*this)(r, diag);
                (*this)(r, diag) = 0.0;
                for (c = diag + 1; c <= Cols(); ++c)
                    (*this)(r, c) -= (*this)(diag, c) * factor;
            }
            // display elimination step:
            cout << "eliminated below pivot in column " << diag << '\n';
            cout << *this;
        }

        // back substitute:
        Matrix soln(Rows(), 1);
        soln(Rows(), 1) = (*this)(Rows(), Cols());   // the last unknown

        for (r = Rows() - 1; r >= 1; --r) {             // the rest
            double sum = 0.0;
            for (diag = r + 1; diag <= Rows(); ++diag)
                sum += (*this)(r, diag) * soln(diag, 1);
            soln(r, 1) = (*this)(r, Cols()) - sum;
        }
        cout << "solution:\n";
        cout << soln;
    }
```

8.3

```
    #include "bitvec.h"

    class EnumSet : public BitVec {
    public:
                EnumSet (const short maxCard) : BitVec(maxCard) {}
                EnumSet (BitVec& v) : BitVec(v)  {*this = (EnumSet&)v;}
        friend  EnumSet  operator +  (EnumSet &s, EnumSet &t);
        friend  EnumSet  operator -  (EnumSet &s, EnumSet &t);
        friend  EnumSet  operator *  (EnumSet &s, EnumSet &t);
        friend  Bool     operator %  (const short elem, EnumSet &s);
        friend  Bool     operator <= (EnumSet &s, EnumSet &t);
        friend  Bool     operator >= (EnumSet &s, EnumSet &t);
        friend  EnumSet& operator << (EnumSet &s, const short elem);
        friend  EnumSet& operator >> (EnumSet &s, const short elem);
    };

    inline EnumSet operator + (EnumSet &s, EnumSet &t)  // union
    {
        return s | t;
    }

    inline EnumSet operator - (EnumSet &s, EnumSet &t)  // difference
    {
        return s & ~t;
    }

    inline EnumSet operator * (EnumSet &s, EnumSet &t)  // intersection
    {
        return s & t;
    }

    inline Bool operator % (const short elem, EnumSet &t)
```



```
{
    return t[elem];
}

inline Bool operator <= (EnumSet &s, EnumSet &t)
{
    return (t & s) == s;
}

inline Bool operator >= (EnumSet &s, EnumSet &t)
{
    return (t & s) == t;
}

EnumSet& operator << (EnumSet &s, const short elem)
{
    s.Set(elem);
    return s;
}

EnumSet& operator >> (EnumSet &s, const short elem)
{
    s.Reset(elem);
    return s;
}
```

8.4

```
typedef int     Key;
typedef double  Data;
enum Bool { false, true };

class Database {
public:
virtual void    Insert  (Key key, Data data)    {}
virtual void    Delete  (Key key)               {}
virtual Bool    Search  (Key key, Data &data)   {return false;}
};
```

A B-tree consists of a set of nodes, where each node may contain up to $2n$ records and have $2n+1$ children. The number n is called the order of the tree. Every node in the tree (except for the root node) must have at least n records. This ensures that at least 50% of the storage capacity is utilized. Furthermore, a nonleaf node that contains m records must have exactly $m+1$ children. The most important property of a B-tree is that the insert and delete operations are designed so that the tree remains balanced at all times.

```
#include <iostream.h>
#include "database.h"

const int maxOrder = 256;       // max tree order

class BTree : public Database {
public:
    class Page;
    class Item {        // represents each stored item
    public:
                Item    (void)      {right = 0;}
                Item    (Key, Data);
        Key&    KeyOf   (void)      {return key;}
        Data&   DataOf  (void)      {return data;}
        Page*&  Subtree (void)      {return right;}
    friend  ostream&    operator << (ostream&, Item&);
```

```
    private:
        Key     key;    // item's key
        Data    data;   // item's data
        Page    *right; // pointer to right subtree
    };

    class Page {        // represents each tree node
    public:
                    Page    (const int size);
                    ~Page   (void)          {delete items;}
        Page*&      Left    (const int ofItem);
        Page*&      Right   (const int ofItem);
        const int   Size    (void)          {return size;}
        int&        Used    (void)          {return used;}
        Item&   operator [] (const int n)   {return items[n];}
        Bool        BinarySearch(Key key, int &idx);
        int         CopyItems   (Page *dest, const int srcIdx,
                                 const int destIdx, const int count);
        Bool        InsertItem  (Item &item, int atIdx);
        Bool        DeleteItem  (int atIdx);
        void        PrintPage   (ostream& os, const int margin);
    private:
        const int   size;   // max no. of items per page
        int         used;   // no. of items on the page
        Page        *left;  // pointer to the left-most subtree
        Item        *items; // the items on the page
    };

public:
                BTree       (const int order);
                ~BTree      (void)      {FreePages(root);}
virtual void    Insert      (Key key, Data data);
virtual void    Delete      (Key key);
virtual Bool    Search      (Key key, Data &data);
friend ostream& operator << (ostream&, BTree&);

protected:
        const int order;// order of tree
        Page    *root;      // root of the tree
        Page    *bufP;  // buffer page for distribution/merging

virtual void    FreePages   (Page *page);
virtual Item*   SearchAux   (Page *tree, Key key);
virtual Item*   InsertAux   (Item *item, Page *page);

virtual void    DeleteAux1  (Key key, Page *page, Bool &underflow);
virtual void    DeleteAux2  (Page *parent,Page *page,
                             const int idx, Bool &underflow);
virtual void    Underflow   (Page *page, Page *child,
                             int idx, Bool &underflow);
};

BTree::Item::Item (Key k, Data d)
{
    key = k;
    data = d;
    right = 0;
}

ostream& operator << (ostream& os, BTree::Item &item)
{
    os << item.key << ' ' << item.data;
    return os;
}
```



```
BTree::Page::Page (const int sz) : size(sz)
{
    used = 0;
    left = 0;
    items = new Item[size];
}

// return the left subtree of an item

BTree::Page*& BTree::Page::Left (const int ofItem)
{
    return ofItem <= 0 ? left: items[ofItem - 1].Subtree();
}

// return the right subtree of an item

BTree::Page*& BTree::Page::Right (const int ofItem)
{
    return ofItem < 0 ? left : items[ofItem].Subtree();
}

// do a binary search on items of a page
// returns true if successful and false otherwise

Bool BTree::Page::BinarySearch (Key key, int &idx)
{
    int low = 0;
    int high = used - 1;
    int mid;

    do {
        mid = (low + high) / 2;
        if (key <= items[mid].KeyOf())
            high = mid - 1;             // restrict to lower half
        if (key >= items[mid].KeyOf())
            low = mid + 1;              // restrict to upper half
    } while (low <= high);

    Bool found = low - high > 1;

    idx = found ? mid : high;
    return found;
}

// copy a set of items from page to page

int BTree::Page::CopyItems (Page *dest, const int srcIdx,
                            const int destIdx, const int count)
{
    for (register i = 0; i < count; ++i)// straight copy
        dest->items[destIdx + i] = items[srcIdx + i];
    return count;
}

// insert an item into a page

Bool BTree::Page::InsertItem (Item &item, int atIdx)
{
    for (register i = used; i > atIdx; --i) // shift right
        items[i] = items[i - 1];
    items[atIdx] = item;                // insert
    return ++used >= size;                  // overflow?
}

// delete an item from a page
```

```
Bool BTree::Page::DeleteItem (int atIdx)
{
    for (register i = atIdx; i < used - 1; ++i)  // shift left
        items[i] = items[i + 1];
    return --used < size/2;                 // underflow?
}

// recursively print a page and its subtrees

void BTree::Page::PrintPage (ostream& os, const int margin)
{
    char margBuf[128];

    // build the margin string:
    for (int i = 0; i <= margin; ++i)
        margBuf[i] = ' ';
    margBuf[i] = '\0';

    // print the left-most child:
    if (Left(0) != 0)
        Left(0)->PrintPage(os, margin + 8);

    // print page and remaining children:
    for (i = 0; i < used; ++i) {
        os << margBuf;
        os << (*this)[i]  << '\n';
        if (Right(i) != 0)
            Right(i)->PrintPage(os, margin + 8);
    }
}

BTree::BTree (const int ord) : order(ord)
{
    root = 0;
    bufP = new Page(2 * order + 2);
}

void BTree::Insert (Key key, Data data)
{
    Item item(key, data), *receive;

    if (root == 0) {                        // empty tree
        root = new Page(2 * order);
        root->InsertItem(item, 0);
    } else if ((receive = InsertAux(&item, root)) != 0) {
        Page *page = new Page(2 * order);   // new root
        page->InsertItem(*receive, 0);
        page->Left(0) = root;
        root = page;
    }
}

void BTree::Delete (Key key)
{
    Bool underflow;

    DeleteAux1(key, root, underflow);
    if (underflow && root->Used() == 0) {   // dispose root
        Page *temp = root;
        root = root->Left(0);
        delete temp;
    }
}
```



```
Bool BTree::Search (Key key, Data &data)
{
    Item *item = SearchAux(root, key);

    if (item == 0)
        return false;
    data = item->DataOf();
    return true;
}

ostream& operator << (ostream& os, BTree &tree)
{
    if (tree.root != 0)
        tree.root->PrintPage(os, 0);
    return os;
}

// recursively free a page and its subtrees

void BTree::FreePages (Page *page)
{
    if (page != 0) {
        FreePages(page->Left(0));
        for (register i = 0; i < page->Used(); ++i)
            FreePages(page->Right(i));
        delete page;
    }
}

// recursively search the tree for an item with matching key

BTree::Item* BTree::SearchAux (Page *tree, Key key)
{
    int     idx;
    Item    *item;

    if (tree == 0)
        return 0;
    if (tree->BinarySearch(key, idx))
        return &((*tree)[idx]);
    return SearchAux(idx < 0 ? tree->Left(0)
                             : tree->Right(idx), key);
}

// insert an item into a page and split the page if it overflows

BTree::Item* BTree::InsertAux (Item *item, Page *page)
{
    Page *child;
    int  idx;

    if (page->BinarySearch(item->KeyOf(), idx))
        return 0;                       // already in tree

    if ((child = page->Right(idx)) != 0)
        item = InsertAux(item, child);      // child is not a leaf

    if (item != 0) {                    // page is a leaf, or passed up
        if (page->Used() < 2 * order) {     // insert in the page
            page->InsertItem(*item, idx + 1);
        } else {                            // page is full, split
            Page *newP = new Page(2 * order);

            bufP->Used() = page->CopyItems(bufP, 0, 0, page->Used());
            bufP->InsertItem(*item, idx + 1);
```

```
            int size = bufP->Used();
            int half = size/2;

            page->Used() = bufP->CopyItems(page, 0, 0, half);
            newP->Used() = bufP->CopyItems(newP, half + 1, 0, size -
half - 1);
            newP->Left(0) = bufP->Right(half);

            *item = (*bufP)[half];      // the mid item
            item->Subtree() = newP;
            return item;
        }
    }
    return 0;
}

// delete an item from a page and deal with underflows

void BTree::DeleteAux1 (Key key, Page *page, Bool &underflow)
{
    int     idx;
    Page    *child;

    underflow = false;
    if (page == 0)
        return;

    if (page->BinarySearch(key, idx)) {
        if ((child = page->Left(idx)) == 0) {    // page is a leaf
            underflow = page->DeleteItem(idx);
        } else {                                 // page is a subtree
            // delete from subtree:
            DeleteAux2(page, child, idx, underflow);
            if (underflow)
                Underflow(page, child, idx - 1, underflow);
        }
    } else {                                     // is not on this page
        child = page->Right(idx);
        DeleteAux1(key, child, underflow);       // should be in child
        if (underflow)
            Underflow(page, child, idx, underflow);
    }
}

// delete an item and deal with underflows by borrowing
// items from neighboring pages or merging two pages

void BTree::DeleteAux2 (Page *parent,Page *page,
                        const int idx, Bool &underflow)
{
    Page *child = page->Right(page->Used() - 1);

    if (child != 0) {                            // page is not a leaf
        DeleteAux2(parent, child, idx, underflow); // go another level
down
        if (underflow)
            Underflow(page, child, page->Used() - 1, underflow);
    } else {                                    // page is a leaf
        // save right:
        Page *right = parent->Right(idx);
        // borrow an item from page for parent:
        page->CopyItems(parent, page->Used() - 1, idx, 1);
        // restore right:
        parent->Right(idx) = right;
```

```
        underflow = page->DeleteItem(page->Used() - 1);
    }
}

// handle underflows

void BTree::Underflow (Page *page, Page *child,
                       int idx, Bool &underflow)
{
    Page *left = idx < page->Used() - 1 ? child : page->Left(idx);
    Page *right = left == child ? page->Right(++idx) : child;

    // copy contents of left, parent item, and right onto bufP:
    int size = left->CopyItems(bufP, 0, 0, left->Used());
    (*bufP)[size] = (*page)[idx];
    bufP->Right(size++) = right->Left(0);
    size += right->CopyItems(bufP, 0, size, right->Used());

    if (size > 2 * order) {
        // distribute bufP items between left and right:
        int half = size/2;
        left->Used() = bufP->CopyItems(left, 0, 0, half);
        right->Used() = bufP->CopyItems(right, half + 1, 0, size - half
- 1);
        right->Left(0) = bufP->Right(half);
        (*page)[idx] = (*bufP)[half];
        page->Right(idx) = right;
        underflow = false;
    } else {
        // merge, and free the right page:
        left->Used() = bufP->CopyItems(left, 0, 0, size);
        underflow = page->DeleteItem(idx);
        delete right;
    }
}
```

A B*-tree is a B-tree in which most nodes are at least 2/3 full (instead of 1/2 full). Instead of splitting a node as soon as it becomes full, an attempt is made to evenly distribute the contents of the node and its neighbor(s) between them. A node is split only when one or both of its neighbors are full too. A B*-tree facilitates more economic utilization of the available store, since it ensures that at least 66% of the storage occupied by the tree is actually used. As a result, the height of the tree is smaller, which in turn improves the search speed. The search and delete operations are exactly as in a B-tree; only the insertion operation is different.

```
class BStar : public BTree {
public:
                BStar       (const int order) : BTree(order) {}
virtual void    Insert      (Key key, Data data);

protected:
virtual Item*   InsertAux   (Item *item, Page *page);
virtual Item*   Overflow    (Item *item, Page *page,
                             Page *child, int idx);
};

// insert with overflow/underflow handling
```

```
void BStar::Insert (Key key, Data data)
{
    Item item(key, data);
    Item *overflow;
    Page *left, *right;
    Bool dummy;

    if (root == 0) {            // empty tree
        root = new Page(2 * order);
        root->InsertItem(item, 0);
    } else if ((overflow = InsertAux(&item, root)) != 0) {
        left = root;            // root becomes a left child
        root = new Page(2 * order);
        right = new Page (2 * order);
        root->InsertItem(*overflow, 0);
        root->Left(0) = left;   // the left-most child of root
        root->Right(0) = right; // the right child of root
        right->Left(0) = overflow->Subtree();
        // right is underflown (size == 0):
        Underflow(root, right, 0, dummy);
    }
}

// inserts and deals with overflows

Item* BStar::InsertAux (Item *item, Page *page)
{
    Page *child;
    int  idx;

    if (page->BinarySearch(item->KeyOf(), idx))
        return 0;                           // already in tree
    if ((child = page->Right(idx)) != 0) {
        // child not a leaf:
        if ((item = InsertAux(item, child)) != 0)
            return Overflow(item, page, child, idx);
    } else if (page->Used() < 2 * order) {  // item fits in node
        page->InsertItem(*item, idx + 1);
    } else {                                // node is full
        int size = page->Used();
        bufP->Used() = page->CopyItems(bufP, 0, 0, size);
        bufP->InsertItem(*item, idx + 1);
        bufP->CopyItems(page, 0, 0, size);
        *item = (*bufP)[size];
        return item;
    }
    return 0;
}

// handles underflows

Item* BStar::Overflow (Item *item, Page *page,
                       Page *child, int idx)
{
    Page *left = idx < page->Used() - 1 ? child : page->Left(idx);
    Page *right = left == child ? page->Right(++idx) : child;

    // copy left, overflown and parent items, and right into buf:
    bufP->Used() = left->CopyItems(bufP, 0, 0, left->Used());
    if (child == left ) {
        bufP->InsertItem(*item, bufP->Used());
        bufP->InsertItem((*page)[idx], bufP->Used());
        bufP->Right(bufP->Used() - 1) = right->Left(0);
        bufP->Used() +=
                right->CopyItems(bufP, 0, bufP->Used(), right->Used());
```

```
    } else {
        bufP->InsertItem((*page)[idx], bufP->Used());
        bufP->Right(bufP->Used() - 1) = right->Left(0);
        bufP->Used() +=
                right->CopyItems(bufP, 0, bufP->Used(), right->Used());
        bufP->InsertItem(*item, bufP->Used());
    }
    if (bufP->Used() < 4 * order + 2) {
        // distribute buf between left and right:
        int size = bufP->Used(), half;

        left->Used() = bufP->CopyItems(left, 0, 0, half = size/2);
        right->Used() = bufP->CopyItems(right, half + 1, 0, size - half
- 1);
        right->Left(0) = bufP->Right(half);
        (*page)[idx] = (*bufP)[half];
        page->Right(idx) = right;
        return 0;
    } else {
        // split int 3 pages:
        Page *newP = new Page(2 * order);
        int  mid1, mid2;

        mid1 = left->Used() = bufP->CopyItems(left, 0, 0, (4 * order +
1) / 3);
        mid2 = right->Used() = bufP->CopyItems(right, mid1 + 1, 0, 4 *
order / 3);
        mid2 += mid1 + 1;
        newP->Used() = bufP->CopyItems(newP, mid2 + 1, 0, (4 * order +
2) / 3);
        right->Left(0) = bufP->Right(mid1);
        bufP->Right(mid1) = right;
        newP->Left(0) = bufP->Right(mid2);
        bufP->Right(mid2) = newP;
        (*page)[idx] = (*bufP)[mid2];
        if (page->Used() < 2 * order) {
            page->InsertItem((*bufP)[mid1], idx);
            return 0;
        } else {
            *item = (*page)[page->Used() - 1];
            (*page)[page->Used() - 1] = (*bufP)[mid1];
            return item;
        }
    }
}
```

9.1

```
template <class Type>
void Swap (Type &x, Type &y)
{
    Type tmp = x;
    x = y;
    y = tmp;
}
```

9.2

```
#include <string.h>

enum Bool {false, true};

template <class Type>
void BubbleSort (Type *names, const int size)
{
    Bool swapped;

    do {
```

```
        swapped = false;
        for (register i = 0; i < size - 1; ++i) {
            if (names[i] > names[i+1]) {
                Type temp = names[i];
                names[i] = names[i+1];
                names[i+1] = temp;
                swapped = true;
            }
        }
    } while (swapped);
}

// specialization:

void BubbleSort (char **names, const int size)
{
    Bool swapped;

    do {
        swapped = false;
        for (register i = 0; i < size - 1; ++i) {
            if (strcmp(names[i], names[i+1]) > 0 ) {
                char *temp = names[i];
                names[i] = names[i+1];
                names[i+1] = temp;
                swapped = true;
            }
        }
    } while (swapped);
}
```

9.3

```
#include <string.h>
#include <iostream.h>

enum Bool {false,true};

typedef char *Str;

template <class Type>
class BinNode {
public:
                BinNode     (const Type&);
                ~BinNode    (void)  {}
    Type&       Value       (void)  {return value;}
    BinNode*&   Left        (void)  {return left;}
    BinNode*&   Right       (void)  {return right;}

    void        FreeSubtree (BinNode *subtree);
    void        InsertNode  (BinNode *node, BinNode *&subtree);
    void        DeleteNode  (const Type&, BinNode *&subtree);
const BinNode*  FindNode    (const Type&, const BinNode *subtree);
    void        PrintNode   (const BinNode *node);

private:
    Type    value;      // node value
    BinNode *left;      // pointer to left child
    BinNode *right;     // pointer to right child
};

template <class Type>
class BinTree {
public:
            BinTree (void);
            ~BinTree(void);
```



```
    void    Insert  (const Type &val);
    void    Delete  (const Type &val);
    Bool    Find    (const Type &val);
    void    Print   (void);

protected:
    BinNode<Type> *root;// root node of the tree
};

template <class Type>
BinNode<Type>::BinNode (const Type &val)
{
    value = val;
    left = right = 0;
}

// specialization:

BinNode<Str>::BinNode (const Str &str)
{
    value = new char[strlen(str) + 1];
    strcpy(value, str);
    left = right = 0;
}

template <class Type>
void BinNode<Type>::FreeSubtree (BinNode<Type> *node)
{
    if (node != 0) {
        FreeSubtree(node->left);
        FreeSubtree(node->right);
        delete node;
    }
}

template <class Type>
void BinNode<Type>::InsertNode (BinNode<Type> *node, BinNode<Type>
*&subtree)
{
    if (subtree == 0)
        subtree = node;
    else if (node->value <= subtree->value)
        InsertNode(node, subtree->left);
    else
        InsertNode(node, subtree->right);
}

// specialization:

void BinNode<Str>::InsertNode (BinNode<Str> *node, BinNode<Str>
*&subtree)
{
    if (subtree == 0)
        subtree = node;
    else if (strcmp(node->value, subtree->value) <= 0)
        InsertNode(node, subtree->left);
    else
        InsertNode(node, subtree->right);
}

template <class Type>
void BinNode<Type>::DeleteNode (const Type &val, BinNode<Type>
*&subtree)
{
    int cmp;
```

```
    if (subtree == 0)
        return;
    if (val < subtree->value)
        DeleteNode(val, subtree->left);
    else if (val > subtree->value)
        DeleteNode(val, subtree->right);
    else {
        BinNode* handy = subtree;
        if (subtree->left == 0)     // no left subtree
            subtree = subtree->right;
        else if (subtree->right == 0)   // no right subtree
            subtree = subtree->left;
        else {                          // left and right subtree
            subtree = subtree->right;
            // insert left subtree into right subtree:
            InsertNode(subtree->left, subtree->right);
        }
        delete handy;
    }
}

// specialization:

void BinNode<Str>::DeleteNode (const Str &str, BinNode<Str> *&subtree)
{
    int cmp;

    if (subtree == 0)
        return;
    if ((cmp = strcmp(str, subtree->value)) < 0)
        DeleteNode(str, subtree->left);
    else if (cmp > 0)
        DeleteNode(str, subtree->right);
    else {
        BinNode<Str>* handy = subtree;
        if (subtree->left == 0)     // no left subtree
            subtree = subtree->right;
        else if (subtree->right == 0)   // no right subtree
            subtree = subtree->left;
        else {                          // left and right subtree
            subtree = subtree->right;
            // insert left subtree into right subtree:
            InsertNode(subtree->left, subtree->right);
        }
        delete handy;
    }
}

template <class Type>
const BinNode<Type>*
BinNode<Type>::FindNode (const Type &val, const BinNode<Type> *subtree)
{
    if (subtree == 0)
        return 0;
    if (val < subtree->value)
        return FindNode(val, subtree->left);
    if (val > subtree->value)
        return FindNode(val, subtree->right);
    return subtree;
}

// specialization:

const BinNode<Str>*
```



```
BinNode<Str>::FindNode (const Str &str, const BinNode<Str> *subtree)
{
    int cmp;

    return (subtree == 0)
            ? 0
            : ((cmp = strcmp(str, subtree->value)) < 0
                ? FindNode(str, subtree->left)
                : (cmp > 0
                    ? FindNode(str, subtree->right)
                    : subtree));

}

template <class Type>
void BinNode<Type>::PrintNode (const BinNode<Type> *node)
{
    if (node != 0) {
        PrintNode(node->left);
        cout << node->value << ' ';
        PrintNode(node->right);
    }
}

template <class Type>
void BinTree<Type>::Insert (const Type &val)
{
    root->InsertNode(new BinNode<Type>(val), root);
}

template <class Type>
BinTree<Type>::BinTree (void)
{
    root = 0;
}

template <class Type>
BinTree<Type>::~BinTree(void)
{
    root->FreeSubtree(root);
}

template <class Type>
void BinTree<Type>::Delete (const Type &val)
{
    root->DeleteNode(val, root);
}

template <class Type>
Bool BinTree<Type>::Find (const Type &val)
{
    return root->FindNode(val, root) != 0;
}

template <class Type>
void BinTree<Type>::Print (void)
{
    root->PrintNode(root);  cout << '\n';
}
```

9.4

```
#include <iostream.h>

enum Bool { false, true };
```

```
template <class Key, class Data>
class Database {
public:
virtual void    Insert  (Key key, Data data)    {}
virtual void    Delete  (Key key)               {}
virtual Bool    Search  (Key key, Data &data)   {return false;}
};

template <class Key, class Data> class Page;

template <class Key, class Data>
class Item {        // represents each stored item
public:
            Item    (void)      {right = 0;}
            Item    (Key, Data);
    Key&    KeyOf   (void)      {return key;}
    Data&   DataOf  (void)      {return data;}
    Page<Key, Data>*&   Subtree (void)      {return right;}
friend  ostream&    operator << (ostream&, Item&);
private:
    Key     key;    // item's key
    Data    data;   // item's data
    Page<Key, Data> *right; // pointer to right subtree
};

template <class Key, class Data>
class Page {        // represents each tree node
public:
                Page    (const int size);
                ~Page   (void)          {delete items;}
    Page*&      Left    (const int ofItem);
    Page*&      Right   (const int ofItem);
    const int   Size    (void)          {return size;}
    int&        Used    (void)          {return used;}
    Item<Key, Data>&    operator [] (const int n)   {return items[n];}
    Bool        BinarySearch(Key key, int &idx);
    int         CopyItems   (Page *dest, const int srcIdx,
                             const int destIdx, const int count);
    Bool        InsertItem  (Item<Key, Data> &item, int atIdx);
    Bool        DeleteItem  (int atIdx);
    void        PrintPage   (ostream& os, const int margin);
private:
    const int   size;   // max no. of items per page
    int         used;   // no. of items on the page
    Page        *left;  // pointer to the left-most subtree
    Item<Key, Data>     *items; // the items on the page
};

template <class Key, class Data>
class BTree : public Database<Key, Data> {
public:
                BTree       (const int order);
                ~BTree      (void)      {FreePages(root);}
virtual void    Insert      (Key key, Data data);
virtual void    Delete      (Key key);
virtual Bool    Search      (Key key, Data &data);
friend ostream& operator << (ostream&, BTree&);

protected:
        const int order;// order of tree
        Page<Key, Data> *root;  // root of the tree
        Page<Key, Data> *bufP;  // buffer page for distribution/merging

virtual void    FreePages   (Page<Key, Data> *page);
virtual Item<Key, Data>* SearchAux   (Page<Key, Data> *tree, Key key);
```



```
virtual Item<Key, Data>* InsertAux   (Item<Key, Data> *item,
                                      Page<Key, Data> *page);

virtual void    DeleteAux1  (Key key, Page<Key, Data> *page,
                             Bool &underflow);
virtual void    DeleteAux2  (Page<Key, Data> *parent,
                             Page<Key, Data> *page,
                             const int idx, Bool &underflow);
virtual void    Underflow   (Page<Key, Data> *page,
                             Page<Key, Data> *child,
                             int idx, Bool &underflow);
};

template <class Key, class Data>
Item<Key, Data>::Item (Key k, Data d)
{
    key = k;
    data = d;
    right = 0;
}

template <class Key, class Data>
ostream& operator << (ostream& os, Item<Key,Data> &item)
{
    os << item.key << ' ' << item.data;
    return os;
}

template <class Key, class Data>
Page<Key, Data>::Page (const int sz) : size(sz)
{
    used = 0;
    left = 0;
    items = new Item<Key, Data>[size];
}

// return the left subtree of an item

template <class Key, class Data>
Page<Key, Data>*& Page<Key, Data>::Left (const int ofItem)
{
    return ofItem <= 0 ? left: items[ofItem - 1].Subtree();
}

// return the right subtree of an item

template <class Key, class Data>
Page<Key, Data>*& Page<Key, Data>::Right (const int ofItem)
{
    return ofItem < 0 ? left : items[ofItem].Subtree();
}

// do a binary search on items of a page
// returns true if successful and false otherwise

template <class Key, class Data>
Bool Page<Key, Data>::BinarySearch (Key key, int &idx)
{
    int low = 0;
    int high = used - 1;
    int mid;

    do {
        mid = (low + high) / 2;
        if (key <= items[mid].KeyOf())
```

```
            high = mid - 1;             // restrict to lower half
        if (key >= items[mid].KeyOf())
            low = mid + 1;              // restrict to upper half
    } while (low <= high);

    Bool found = low - high > 1;

    idx = found ? mid : high;
    return found;
}

// copy a set of items from page to page

template <class Key, class Data>
int Page<Key, Data>::CopyItems (Page<Key, Data> *dest, const int
srcIdx,
                                const int destIdx, const int count)
{
    for (register i = 0; i < count; ++i)// straight copy
        dest->items[destIdx + i] = items[srcIdx + i];
    return count;
}

// insert an item into a page

template <class Key, class Data>
Bool Page<Key, Data>::InsertItem (Item<Key, Data> &item, int atIdx)
{
    for (register i = used; i > atIdx; --i) // shift right
        items[i] = items[i - 1];
    items[atIdx] = item;                // insert
    return ++used >= size;                  // overflow?
}

// delete an item from a page

template <class Key, class Data>
Bool Page<Key, Data>::DeleteItem (int atIdx)
{
    for (register i = atIdx; i < used - 1; ++i)  // shift left
        items[i] = items[i + 1];
    return --used < size/2;                 // underflow?
}

// recursively print a page and its subtrees

template <class Key, class Data>
void Page<Key, Data>::PrintPage (ostream& os, const int margin)
{
    char margBuf[128];

    // build the margin string:
    for (int i = 0; i <= margin; ++i)
        margBuf[i] = ' ';
    margBuf[i] = '\0';

    // print the left-most child:
    if (Left(0) != 0)
        Left(0)->PrintPage(os, margin + 8);

    // print page and remaining children:
    for (i = 0; i < used; ++i) {
        os << margBuf;
        os << (*this)[i]  << '\n';
        if (Right(i) != 0)
```



```
            Right(i)->PrintPage(os, margin + 8);
    }
}

template <class Key, class Data>
BTree<Key, Data>::BTree (const int ord) : order(ord)
{
    root = 0;
    bufP = new Page<Key, Data>(2 * order + 2);
}

template <class Key, class Data>
void BTree<Key, Data>::Insert (Key key, Data data)
{
    Item<Key, Data> item(key, data), *receive;

    if (root == 0) {                        // empty tree
        root = new Page<Key, Data>(2 * order);
        root->InsertItem(item, 0);
    } else if ((receive = InsertAux(&item, root)) != 0) {
        Page<Key, Data> *page = new Page<Key, Data>(2 * order);  // new
root
        page->InsertItem(*receive, 0);
        page->Left(0) = root;
        root = page;
    }
}

template <class Key, class Data>
void BTree<Key, Data>::Delete (Key key)
{
    Bool underflow;

    DeleteAux1(key, root, underflow);
    if (underflow && root->Used() == 0) {   // dispose root
        Page<Key, Data> *temp = root;
        root = root->Left(0);
        delete temp;
    }
}

template <class Key, class Data>
Bool BTree<Key, Data>::Search (Key key, Data &data)
{
    Item<Key, Data> *item = SearchAux(root, key);

    if (item == 0)
        return false;
    data = item->DataOf();
    return true;
}

template <class Key, class Data>
ostream& operator << (ostream& os, BTree<Key, Data> &tree)
{
    if (tree.root != 0)
        tree.root->PrintPage(os, 0);
    return os;
}

// recursively free a page and its subtrees

template <class Key, class Data>
void BTree<Key, Data>::FreePages (Page<Key, Data> *page)
{
```

```
    if (page != 0) {
        FreePages(page->Left(0));
        for (register i = 0; i < page->Used(); ++i)
            FreePages(page->Right(i));
        delete page;
    }
}

// recursively search the tree for an item with matching key

template <class Key, class Data>
Item<Key, Data>* BTree<Key, Data>::
        SearchAux (Page<Key, Data> *tree, Key key)
{
    int     idx;
    Item<Key, Data> *item;

    if (tree == 0)
        return 0;
    if (tree->BinarySearch(key, idx))
        return &((*tree)[idx]);
    return SearchAux(idx < 0 ? tree->Left(0)
                             : tree->Right(idx), key);
}

// insert an item into a page and split the page if it overflows

template <class Key, class Data>
Item<Key, Data>* BTree<Key, Data>::InsertAux (Item<Key, Data> *item,
                                    Page<Key, Data> *page)
{
    Page<Key, Data> *child;
    int  idx;

    if (page->BinarySearch(item->KeyOf(), idx))
        return 0;                       // already in tree

    if ((child = page->Right(idx)) != 0)
        item = InsertAux(item, child);      // child is not a leaf

    if (item != 0) {                    // page is a leaf, or passed up
        if (page->Used() < 2 * order) {     // insert in the page
            page->InsertItem(*item, idx + 1);
        } else {                            // page is full, split
            Page<Key, Data> *newP = new Page<Key, Data>(2 * order);

            bufP->Used() = page->CopyItems(bufP, 0, 0, page->Used());
            bufP->InsertItem(*item, idx + 1);

            int size = bufP->Used();
            int half = size/2;

            page->Used() = bufP->CopyItems(page, 0, 0, half);
            newP->Used() = bufP->CopyItems(newP, half + 1, 0, size -
half - 1);
            newP->Left(0) = bufP->Right(half);

            *item = (*bufP)[half];      // the mid item
            item->Subtree() = newP;
            return item;
        }
    }
    return 0;
}
```



```
// delete an item from a page and deal with underflows

template <class Key, class Data>
void BTree<Key, Data>::DeleteAux1 (Key key,
                        Page<Key, Data> *page, Bool &underflow)
{
    int     idx;
    Page<Key, Data> *child;

    underflow = false;
    if (page == 0)
        return;

    if (page->BinarySearch(key, idx)) {
        if ((child = page->Left(idx)) == 0) {    // page is a leaf
            underflow = page->DeleteItem(idx);
        } else {                                 // page is a subtree
            // delete from subtree:
            DeleteAux2(page, child, idx, underflow);
            if (underflow)
                Underflow(page, child, idx - 1, underflow);
        }
    } else {                                     // is not on this page
        child = page->Right(idx);
        DeleteAux1(key, child, underflow);       // should be in child
        if (underflow)
            Underflow(page, child, idx, underflow);
    }
}

// delete an item and deal with underflows by borrowing
// items from neighboring pages or merging two pages

template <class Key, class Data>
void BTree<Key, Data>::DeleteAux2 (Page<Key, Data> *parent,
                        Page<Key, Data> *page,
                        const int idx, Bool &underflow)
{
    Page<Key, Data> *child = page->Right(page->Used() - 1);

    if (child != 0) {                            // page is not a leaf
        DeleteAux2(parent, child, idx, underflow); // go another level
down
        if (underflow)
            Underflow(page, child, page->Used() - 1, underflow);
    } else {                                    // page is a leaf
        // save right:
        Page<Key, Data> *right = parent->Right(idx);
        // borrow an item from page for parent:
        page->CopyItems(parent, page->Used() - 1, idx, 1);
        // restore right:
        parent->Right(idx) = right;
        underflow = page->DeleteItem(page->Used() - 1);
    }
}

// handle underflows

template <class Key, class Data>
void BTree<Key, Data>::Underflow (Page<Key, Data> *page,
                        Page<Key, Data> *child,
                        int idx, Bool &underflow)
{
    Page<Key, Data> *left =
            idx < page->Used() - 1 ? child : page->Left(idx);
```

```
    Page<Key, Data> *right =
            left == child ? page->Right(++idx) : child;

    // copy contents of left, parent item, and right onto bufP:
    int size = left->CopyItems(bufP, 0, 0, left->Used());
    (*bufP)[size] = (*page)[idx];
    bufP->Right(size++) = right->Left(0);
    size += right->CopyItems(bufP, 0, size, right->Used());

    if (size > 2 * order) {
        // distribute bufP items between left and right:
        int half = size/2;
        left->Used() = bufP->CopyItems(left, 0, 0, half);
        right->Used() = bufP->CopyItems(right, half + 1, 0, size - half
- 1);
        right->Left(0) = bufP->Right(half);
        (*page)[idx] = (*bufP)[half];
        page->Right(idx) = right;
        underflow = false;
    } else {
        // merge, and free the right page:
        left->Used() = bufP->CopyItems(left, 0, 0, size);
        underflow = page->DeleteItem(idx);
        delete right;
    }
}

//-------------------------------------------------------------

template <class Key, class Data>
class BStar : public BTree<Key, Data> {
public:
                BStar       (const int order) : BTree<Key, Data>(order)
{}
virtual void    Insert      (Key key, Data data);

protected:
virtual Item<Key, Data>* InsertAux (Item<Key, Data> *item,
                                    Page<Key, Data> *page);
virtual Item<Key, Data>* Overflow  (Item<Key, Data> *item,
                                    Page<Key, Data> *page,
                                    Page<Key, Data> *child, int
idx);
};

// insert with overflow/underflow handling

template <class Key, class Data>
void BStar<Key, Data>::Insert (Key key, Data data)
{
    Item<Key, Data> item(key, data);
    Item<Key, Data> *overflow;
    Page<Key, Data> *left, *right;
    Bool dummy;

    if (root == 0) {            // empty tree
        root = new Page<Key, Data>(2 * order);
        root->InsertItem(item, 0);
    } else if ((overflow = InsertAux(&item, root)) != 0) {
        left = root;            // root becomes a left child
        root = new Page<Key, Data>(2 * order);
        right = new Page<Key, Data>(2 * order);
        root->InsertItem(*overflow, 0);
        root->Left(0) = left;   // the left-most child of root
        root->Right(0) = right; // the right child of root
```



```
        right->Left(0) = overflow->Subtree();
        // right is underflown (size == 0):
        Underflow(root, right, 0, dummy);
    }
}

// inserts and deals with overflows

template <class Key, class Data>
Item<Key, Data>* BStar<Key, Data>::InsertAux (Item<Key, Data> *item,
                                    Page<Key, Data> *page)
{
    Page<Key, Data> *child;
    int  idx;

    if (page->BinarySearch(item->KeyOf(), idx))
        return 0;                               // already in tree
    if ((child = page->Right(idx)) != 0) {
        // child not a leaf:
        if ((item = InsertAux(item, child)) != 0)
            return Overflow(item, page, child, idx);
    } else if (page->Used() < 2 * order) {  // item fits in node
        page->InsertItem(*item, idx + 1);
    } else {                                    // node is full
        int size = page->Used();
        bufP->Used() = page->CopyItems(bufP, 0, 0, size);
        bufP->InsertItem(*item, idx + 1);
        bufP->CopyItems(page, 0, 0, size);
        *item = (*bufP)[size];
        return item;
    }
    return 0;
}

// handles underflows

template <class Key, class Data>
Item<Key, Data>* BStar<Key, Data>::Overflow (Item<Key, Data> *item,
                        Page<Key, Data> *page,
                        Page<Key, Data> *child, int idx)
{
    Page<Key, Data> *left =
                idx < page->Used() - 1 ? child : page->Left(idx);
    Page<Key, Data> *right =
                left == child ? page->Right(++idx) : child;

    // copy left, overflown and parent items, and right into buf:
    bufP->Used() = left->CopyItems(bufP, 0, 0, left->Used());
    if (child == left ) {
        bufP->InsertItem(*item, bufP->Used());
        bufP->InsertItem((*page)[idx], bufP->Used());
        bufP->Right(bufP->Used() - 1) = right->Left(0);
        bufP->Used() +=
                right->CopyItems(bufP, 0, bufP->Used(), right->Used());
    } else {
        bufP->InsertItem((*page)[idx], bufP->Used());
        bufP->Right(bufP->Used() - 1) = right->Left(0);
        bufP->Used() +=
                right->CopyItems(bufP, 0, bufP->Used(), right->Used());
        bufP->InsertItem(*item, bufP->Used());
    }
    if (bufP->Used() < 4 * order + 2) {
        // distribute buf between left and right:
        int size = bufP->Used(), half;
```

```
            left->Used() = bufP->CopyItems(left, 0, 0, half = size/2);
            right->Used() = bufP->CopyItems(right, half + 1, 0, size - half
- 1);
            right->Left(0) = bufP->Right(half);
            (*page)[idx] = (*bufP)[half];
            page->Right(idx) = right;
            return 0;
        } else {
            // split int 3 pages:
            Page<Key, Data> *newP = new Page<Key, Data>(2 * order);
            int  mid1, mid2;

            mid1 = left->Used() = bufP->CopyItems(left, 0, 0, (4 * order +
1) / 3);
            mid2 = right->Used() = bufP->CopyItems(right, mid1 + 1, 0, 4 *
order / 3);
            mid2 += mid1 + 1;
            newP->Used() = bufP->CopyItems(newP, mid2 + 1, 0, (4 * order +
2) / 3);
            right->Left(0) = bufP->Right(mid1);
            bufP->Right(mid1) = right;
            newP->Left(0) = bufP->Right(mid2);
            bufP->Right(mid2) = newP;
            (*page)[idx] = (*bufP)[mid2];
            if (page->Used() < 2 * order) {
                page->InsertItem((*bufP)[mid1], idx);
                return 0;
            } else {
                *item = (*page)[page->Used() - 1];
                (*page)[page->Used() - 1] = (*bufP)[mid1];
                return item;
            }
        }
    }
```

10.1

```
    enum PType  {controlPack, dataPack, diagnosePack};
    enum Bool   {false, true};

    class Packet {
    public:
        //...
        PType   Type    (void)  {return dataPack;}
        Bool    Valid   (void)  {return true;}
    };

    class Connection {
    public:
        //...
        Bool    Active  (void)  {return true;}
    };

    class InactiveConn  {};
    class InvalidPack   {};
    class UnknownPack   {};

    void ReceivePacket (Packet *pack, Connection *c)
        throw(InactiveConn, InvalidPack, UnknownPack)
    {
        if (!c->Active())
            throw InactiveConn();
        if (!pack->Valid())
            throw InvalidPack();

        switch (pack->Type()) {
```



```
        case controlPack:   //...
                                break;
        case dataPack:      //...
                                break;
        case diagnosePack:  //...
                                break;
        default:            //...
            throw UnknownPack();
    }
}
```

10.2

```
#include <iostream.h>

class DimsDontMatch {};
class BadDims       {};
class BadRow        {};
class BadCol        {};
class HeapExhausted {};

class Matrix {
public:
                Matrix       (const short rows, const short cols);
                Matrix       (const Matrix&);
                ~Matrix      (void)             {delete elems;}
        double& operator ()  (const short row, const short col);
        Matrix& operator =   (const Matrix&);

friend ostream& operator <<  (ostream&, Matrix&);
friend  Matrix  operator  +  (Matrix&, Matrix&);
friend  Matrix  operator  -  (Matrix&, Matrix&);
friend  Matrix  operator  *  (Matrix&, Matrix&);
        const short Rows     (void)     {return rows;}
        const short Cols     (void)     {return cols;}

private:
        const short rows;     // matrix rows
        const short cols;     // matrix columns
        double      *elems;   // matrix elements
};


Matrix::Matrix (const short r, const short c) : rows(r), cols(c)
{
    if (rows <= 0 || cols <= 0)
        throw BadDims();
    elems = new double[rows * cols];
    if (elems == 0)
        throw HeapExhausted();
}

Matrix::Matrix (const Matrix &m) : rows(m.rows), cols(m.cols)
{
    int n = rows * cols;
    if (rows <= 0 || cols <= 0)
        throw BadDims();
    elems = new double[n];
    if (elems == 0)
        throw HeapExhausted();
    for (register i = 0; i < n; ++i)    // copy elements
        elems[i] = m.elems[i];
}

double& Matrix::operator () (const short row, const short col)
{
```

```
    if (row <= 0 || row > rows)
        throw BadRow();
    if (col <= 0 || col > cols)
        throw BadCol();
    return elems[(row - 1)*cols + (col - 1)];
}

Matrix& Matrix::operator = (const Matrix &m)
{
    if (rows == m.rows && cols == m.cols) {     // must match
        int n = rows * cols;
        for (register i = 0; i < n; ++i)    // copy elements
            elems[i] = m.elems[i];
    } else
        throw DimsDontMatch();
    return *this;
}

ostream& operator << (ostream &os, Matrix &m)
{
    for (register r = 1; r <= m.rows; ++r) {
        for (int c = 1; c <= m.cols; ++c)
            os << m(r,c) << '\t';
        os << '\n';
    }
    return os;
}

Matrix operator + (Matrix &p, Matrix &q)
{
    if (p.rows != q.rows || p.cols != q.cols)
        throw DimsDontMatch();
    Matrix m(p.rows, p.cols);
    if (p.rows == q.rows && p.cols == q.cols)
        for (register r = 1; r <= p.rows; ++r)
            for (register c = 1; c <= p.cols; ++c)
                m(r,c) = p(r,c) + q(r,c);
    return m;
}

Matrix operator - (Matrix &p, Matrix &q)
{
    if (p.rows != q.rows || p.cols != q.cols)
        throw DimsDontMatch();
    Matrix m(p.rows, p.cols);
    if (p.rows == q.rows && p.cols == q.cols)
        for (register r = 1; r <= p.rows; ++r)
            for (register c = 1; c <= p.cols; ++c)
                m(r,c) = p(r,c) - q(r,c);
    return m;
}

Matrix operator * (Matrix &p, Matrix &q)
{
    if (p.cols != q.rows)
        throw DimsDontMatch();
    Matrix m(p.rows, q.cols);
    if (p.cols == q.rows)
        for (register r = 1; r <= p.rows; ++r)
            for (register c = 1; c <= q.cols; ++c) {
                m(r,c) = 0.0;
                for (register i = 1; i <= p.cols; ++i)
                    m(r,c) += p(r,c) * q(r,c);
            }
    return m;
```



```
}
```