# SYLLABUS

## OBJECT ORIENTED PROGRAMMING WITH C++
## (Common to CSE & ISE)

**Subject Code: 10CS36** **I.A. Marks : 25**
**Hours/Week : 04** **Exam Hours: 03**
**Total Hours : 52** **Exam Marks: 100**

### PART – A

**UNIT 1** **6 Hours**

**Introduction:** Overview of C++, Sample C++ program, Different data types,operators, expressions, and statements, arrays and strings, pointers & userdefined types Function Components, argument passing, inline functions, function overloading, recursive functions

**UNIT 2** **7 Hours**

**Classes & Objects – I:** Class Specification, Class Objects, Scope resolution operator, Access members, Defining member functions, Data hiding, Constructors, Destructor s, Parameterized constructors, Static data members, Functions

**UNIT 3** **7 Hours**

**Classes & Objects –II:** Friend functions, Passing objects as arguments, Returning objects, Arrays of objects, Dynamic objects, Pointers to objects, Copy constructors, Generic functions and classes, Applications Operator overloading using friend functions such as +, - , pre-increment, post-increment, [ ] etc., overloading <<, >>.

**UNIT 4** **6 Hours**

**Inheritance – I:** Base Class, Inheritance and protected members, Protected base class inheritance, Inheriting multiple base classes

### PART – B

**UNIT 5** **6 Hours**

**Inheritance – II:** Constructors, Destructors and Inheritance, Passing parameters to base class constructors, Granting access, Virtual base classes

**UNIT 6** **7 Hours**

**Virtual functions, Polymorphism:** Virtual function, Calling a Virtual function through a base class reference, Virtual attribute is inherited, Virtual functions are hierarchical, Pure virtual functions, Abstract classes, Using virtual functions, Early and late binding.

**UNIT 7** **6 Hours**

**I/O System Basics, File I/0:** C++ stream classes, Formatted I/O, I/O manipulators, fstream and the File classes, File operations

**UNIT 8** **7 Hours**

**Exception Handling, STL:** Exception handling fundamentals, Exception handling options STL: An overview, containers, vectors, lists, maps.

**Text Books:**

1. Herbert Schildt: The Complete Reference C++, 4th Edition, Tata McGraw Hill, 2003.

**Reference Books:**

1. Stanley B.Lippmann, Josee Lajore: C++ Primer, 4th Edition, Pearson Education, 2005.
2. Paul J Deitel, Harvey M Deitel: C++ for Programmers, Pearson Education, 2009.
3. K R Venugopal, Rajkumar Buyya, T Ravi Shankar: Mastering C++, Tata McGraw Hill, 1999.

# INDEX

# UNIT 1
# Introduction

**1.1 Overview of C++**

**1.2 Sample C++ program**

**1.3 Different data types**

**1.4 Operators, expressions, and statements**

**1.5 Arrays and strings**

**1.6 Pointers & user-defined types**

**1.7 Function Components**

**1.8 Argument passing**

**1.9 Inline functions**

**1.10 Function overloading**

**1.11 Recursive functions**

## 1.1 Overview of C++

C++ began as an expanded version of C. The C++ extensions were first invented by Bjarne Stroustrup in 1979 at Bell Laboratories in Murray Hill, New Jersey. He initially called the new language "C with Classes." However, in 1983 the name was changed to C++. Since C++ was first invented, it has undergone three major revisions, with each adding to and altering the language. The first revision was in 1985 and the second in 1990. The third occurred during the standardization of C++. Several years ago, work began on a standard for C++. Toward that end, a joint ANSI (American National Standards Institute) and ISO (International Standards Organization) standardization committee was formed. The first draft of the proposed standard was created on January 25, 1994.

The ANSI/ISO C++ committee kept the features first defined by Stroustrup and added some new ones as well. But in general, this initial draft reflected the state of C++ at the time.

**What Is Object-Oriented Programming?**

Since object-oriented programming (OOP) drove the creation of C++, it is necessary to understand its foundational principles. OOP is a powerful way to approach the job of programming. Programming methodologies have changed dramatically since the invention of the computer, primarily to accommodate the increasing complexity of programs. For example, when computers were first invented, programming was done by toggling in the binary machine instructions using the computer's front panel. As long as programs were just a few hundred instructions long, this approach worked. As programs grew, assembly language was invented so that a programmer could deal with larger, increasingly complex programs, using symbolic representations of the machine instructions. As programs continued to grow, high-level languages were introduced that gave the programmer more tools with which to handle complexity. The first widespread language was, of course, FORTRAN. Although FORTRAN was a very impressive first step, it is hardly a language that encourages clear, easy-tounderstand programs.

The 1960s gave birth to structured programming. This is the method encouraged by languages such asCand Pascal. The use of structured languages made it possible to write moderately

complex programs fairly easily. Structured languages are characterized by their support for stand-alone subroutines, local variables, rich control constructs, and their lack of reliance upon theGOTO. Although structured languages are a powerful tool, they reach their limit when a project becomes too large.

Object-oriented programming took the best ideas of structured programming and combined them with several new concepts. The result was a different way of organizing a program. In the most general sense, a program can be organized in one of two ways: around its code (what is happening) or around its data (who is being affected). Using only structured programming techniques, programs are typically organized around code. This approach can be thought of as "code acting on data." For example, a program written in a structured language such as C is defined by its functions, any of which may operate on any type of data used by the program.
Object-oriented programs work the other way around. They are organized around data, with the key principle being "data controlling access to code." In an object-oriented language, you define the data and the routines that are permitted to act on that data. Thus, a data type defines precisely what sort of operations can be applied to that data.
To support the principles of object-oriented programming, all OOP languages have three traits in common: encapsulation, polymorphism, and inheritance. Let's examine each.

**Encapsulation**

*Encapsulation* is the mechanism that binds together code and the data it manipulates, and keeps both safe from outside interference and misuse. In an object-oriented language, code and data may be combined in such a way that a self-contained "black box" is created. When code and data are linked together in this fashion, an *object* is created. In other words, an object is the device that supports encapsulation. Within an object, code, data, or both may be *private* to that object or *public*. Private code or data is known to and accessible only by another part of the object. That is, private code or data may not be accessed by a piece of the program that exists outside the object. When code or data is public, other parts of your program may access it even though it is defined within an object. Typically, the public parts of an object are used to provide a controlled interface to the private elements of the object. For all intents and purposes, an object is a variable of a user-defined type. It may seem strange that an object that links both

code and data can be thought of as a variable. However, in object-oriented programming, this is precisely the case. Each time you define a new type of object, you are creating a new data type. Each specific instance of this data type is a compound variable.

**Polymorphism**

Object-oriented programming languages support *polymorphism*, which is characterized by the phrase "one interface, multiple methods." In simple terms, polymorphism is the attribute that allows one interface to control access to a general class of actions. The specific action selected is determined by the exact nature of the situation. A real-world example of polymorphism is a thermostat. No matter what type of furnace your house has (gas, oil, electric, etc.), the thermostat works the same way. In this case, the thermostat (which is the interface) is the same no matter what type of furnace (method) you have. For example, if you want a 70-degree temperature, you set the thermostat to 70 degrees. It doesn't matter what type of furnace actually provides the heat. This same principle can also apply to programming.

For example, you might have a program that defines three different types of stacks. One stack is used for integer values, one for character values, and one for floating-point values. Because of polymorphism, you can define one set of names, **push( )** and **pop( )**, that can be used for all three stacks. In your program you will create three specific versions of these functions, one for each type of stack, but names of the functions will be the same. The compiler will automatically select the right function based upon the data being stored. Thus, the interface to a stack—the functions **push( )** and **pop( )**—are the same no matter which type of stack is being used. The individual versions of these functions define the specific implementations (methods) for each type of data. Polymorphism helps reduce complexity by allowing the same interface to be used to access a general class of actions. It is the compiler's job to select the *specific action* (i.e., method) as it applies to each situation. You, the programmer, don't need to do this selection manually. You need only remember and utilize the *general interface*. The first object-oriented programming languages were interpreters, so polymorphism was, of course, supported at run time. However, C++ is a compiled language. Therefore, in C++, both run-time and compile-time polymorphism are supported.

**Inheritance**

*Inheritance* is the process by which one object can acquire the properties of another object. This is important because it supports the concept of *classification*. If you think about it, most knowledge is made manageable by hierarchical classifications. For example, a Red Delicious apple is part of the classification *apple*, which in turn is part of the *fruit* class, which is under the larger class *food*. Without the use of classifications, each object would have to define explicitly all of its characteristics. However, through the use of classifications, an object need only define those qualities that make it unique within its class. It is the inheritance mechanism that makes it possible for one object to be a specific instance of a more general case. As you will see, inheritance is an important aspect of object-oriented programming

## 1.2 A Sample C++ Program

Let's start with the short sample C++ program shown here.

```
#include <iostream>
using namespace std;
int main()
{
int i;
cout << "This is output.\n"; // this is a single line comment
/* you can still use C style comments */
// input a number using >>
cout << "Enter a number: ";
cin >> i;
// now, output a number using <<
cout << i << " squared is " << i*i << "\n";
return 0;
}
```

As you can see, this program looks much different from the C subset programs found in Part One. A line-by-line commentary will be useful. To begin, the header **<iostream>** is included. This header supports C++-style I/O operations. (**<iostream>** is to C++ what **stdio.h** is to C.) Notice one other thing: there is no **.h** extension to the name **iostream**. The reason is that

**<iostream>** is one of the modern-style headers defined by Standard C++. Modern C++ headers do not use the **.h** extension.

The next line in the program is

using namespace std;

This tells the compiler to use the **std** namespace. Namespaces are a recent addition to C++. Anamespace creates a declarative region in which various program elements can be placed. Namespaces help in the organization of large programs. The **using** statement informs the compiler that you want to use the **std** namespace. This is the namespace in which the entire Standard C++ library is declared. By using the **std** namespace you simplify access to the standard library. The programs in Part One, which use only the C subset, don't need a namespace statement because the C library functions are also available in the default, global namespace.

*Since both new-style headers and namespaces are recent additions to C++, you mayencounter older code that does not use them. Also, if you are using an older compiler,it may not support them. Instructions for using an older compiler are found later in this chapter.*

Now examine the following line.

int main()

Notice that the parameter list in **main( )** is empty. In C++, this indicates that **main( )** has no parameters. This differs from C. In C, a function that has no parameters must use **void** in its parameter list, as shown here:

int main(void)

This was the way **main( )** was declared in the programs in Part One. However, in C++, the use of **void** is redundant and unnecessary. As a general rule, in C++ when a function takes no parameters, its parameter list is simply empty; the use of **void** is not required.

The next line contains two C++ features.

cout << "This is output.\n"; // this is a single line comment

First, the statement

cout << "This is output.\n";

Assuming that **i** has the value 10, this statement causes the phrase **10 squared is 100** to be displayed, followed by a carriage return-linefeed. As this line illustrates, you can run together several **<<** output operations.

The program ends with this statement:

return 0;

This causes zero to be returned to the calling process (which is usually the operating system). This works the same in C++ as it does in C. Returning zero indicates that the program terminated normally. Abnormal program termination should be signaled by returning a nonzero value. You may also use the values **EXIT_SUCCESS** and **EXIT_FAILURE** if you like.

## 1.3 Different data types

**The Five Basic Data Types**

There are five atomic data types in the C subset: character, integer, floating-point, double floating-point, and valueless (**char**, **int**, **float**, **double**, and **void**, respectively). As you will see, all other data types in C are based upon one of these types. The size and range of these data types may vary between processor types and compilers. However, in all cases a character is 1 byte. The size of an integer is usually the same as the word length of the execution environment of the program. For most 16-bit environments, such as DOS or Windows 3.1, an integer is 16 bits. For most 32-bit environments, such as Windows 2000, an integer is 32 bits. However, you cannot make assumptions about the size of an integer if you want your programs to be portable to the widest range of environments. It is important to understand that both C and C++ only stipulate the *minimal range* of each data type, not its size in bytes.

*To the five basic data types defined by C, C++ adds two more:* ***bool*** *and* ***wchar_t****. These are discussed in Part Two.* The exact format of floating-point values will depend upon how they are implemented. Integers will generally correspond to the natural size of a word on the host computer. Values of type **char** are generally used to hold values defined by the ASCII character set. Values outside that range may be handled differently by different compilers. The range of **float** and **double** will depend upon the method used to represent the floating-point numbers. Whatever the method, the range is quite large. Standard C specifies that the minimum range for a floating-point value is 1E−37 to 1E+37. The minimum number of digits of precision for each floating-point type is shown in Table 1-1.

*Standard C++ does not specify a minimum size or range for the basic types. Instead, it simply states that they must meet certain requirements. For example, Standard C++states that an* ***int***

*will "have the natural size suggested by the architecture of the execution environment." In all cases, this will meet or exceed the minimum ranges specified by Standard C. Each C++ compiler specifies the size and range of the basic types in the header* ***<climits>****.*

| **Type** | **Typical Size in Bits** | **Minimal Range** |
|---|---|---|
| char | 8 | -127 to 127 |
| unsigned char | 8 | 0 to 255 |
| signed char | 8 | -127 to 127 |
| int | 16 or 32 | -32,767 to 32,767 |
| unsigned int | 16 or 32 | 0 to 65,535 |
| signed int | 16 or 32 | same as **int** |
| short int | 16 | -32,767 to 32,767 |
| unsigned short int | 16 | 0 to 65,535 |
| signed short int | 16 | same as **short int** |
| long int | 32 | -2,147,483,647 to 2,147,483,647 |
| signed long int | 32 | same as **long int** |
| unsigned long int | 32 | 0 to 4,294,967,295 |
| float | 32 | Six digits of precision |
| double | 64 | Ten digits of precision |
| long double | 80 | Ten digits of precision |

Table 1-1. All Data Types Defined by the ANSI/ISO C Standard

**Modifying the Basic Types**

Except for type **void**, the basic data types may have various modifiers preceding them. You use a *modifier* to alter the meaning of the base type to fit various situations more precisely. The list of modifiers is shown here:

signed

unsigned

long

short

You can apply the modifiers **signed**, **short**, **long**, and **unsigned** to integer base types.You can apply **unsigned** and **signed** to characters. You may also apply **long** to **double**. Table 1-1 shows all valid data type combinations, along with their minimal ranges and approximate bit widths. (These values also apply to a typical C++ implementation.) Remember, the table shows the *minimum range* that these types will have as specified by Standard C/C++, not their typical range. For example, on computers that use two's complement arithmetic (which is nearly all), an integer will have a range of at least 32,767 to –32,768. The use of **signed** on integers is allowed, but redundant because the default integer declaration assumes a signed number. The most important use of **signed** is to modify **char** in implementations in which **char** is unsigned by default. The difference between signed and unsigned integers is in the way that the highorder bit of the integer is interpreted. If you specify a signed integer, the compiler generates code that assumes that the high-order bit of an integer is to be used as a *sign flag*. If the sign flag is 0, the number is positive; if it is 1, the number is negative.

In general, negative numbers are represented using the *two's complement* approach, which reverses all bits in the number (except the sign flag), adds 1 to this number, and sets the sign flag to 1. Signed integers are important for a great many algorithms, but they only have half the absolute magnitude of their unsigned relatives. For example, here is 32,767: 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 If the high-order bit were set to 1, the number would be interpreted as −1. However, if you declare this to be an **unsigned int**, the number becomes 65,535 when the highorder bit is set to 1. When a type modifier is used by itself (that is, when it does not precede a basic type), then **int** is assumed.

Thus, the following sets of type specifiers are equivalent:

| Specifier | Same As |
|---|---|
| signed | signed int |
| unsigned | unsigned int |
| long | long int |
| short | short int |

Although the **int** is implied, many programmers specify the **int** anyway.

## 1.4 Operators, expressions, and statements

### Operators

C/C++ is rich in built-in operators. In fact, it places more significance on operators than do most other computer languages. There are four main classes of operators: *arithmetic,relational*, *logical*, and *bitwise*. In addition, there are some special operators for particular tasks.

### The Assignment Operator

You can use the assignment operator within any valid expression. This is not the case with many computer languages (including Pascal, BASIC, and FORTRAN), which treat the assignment operator as a special case statement. The general form of the assignment operator is

*variable_name = expression;*

where an expression may be as simple as a single constant or as complex as you require. C/C++ uses a single equal sign to indicate assignment (unlike Pascal or Modula-2, which use the := construct). The *target*, or left part, of the assignment must be a variable or a pointer, not a function or a constant. Frequently in literature on C/C++ and in compiler error messages you will see these two terms: lvalue and rvalue. Simply put, an *lvalue* is any object that can occur on the left side of an assignment statement. For all practical purposes, "lvalue" means "variable." The term *rvalue* refers to expressions on the right side of an assignment and simply means the value of an expression.

### Type Conversion in Assignments

When variables of one type are mixed with variables of another type, a *type conversion* will occur. In an assignment statement, the type conversion rule is easy: The value of the right side

(expression side) of the assignment is converted to the type of the left side (target variable), as illustrated here:

```
int x;
char ch;
float f;
void func(void)
{
ch = x; /* line 1 */
x = f; /* line 2 */
f = ch; /* line 3 */
f = x; /* line 4 */
}
```

**Multiple Assignments**

C/C++ allows you to assign many variables the same value by using multiple assignments in a single statement. For example, this program fragment assigns **x**, **y**, and **z** the value 0: x = y = z = 0; In professional programs, variables are frequently assigned common values using this method.

**Arithmetic Operators**

Table 2-4 lists C/C++'s arithmetic operators. The operators +, −, *, and / work as they do in most other computer languages. You can apply them to almost any built-in data type. When you apply / to an integer or character, any remainder will be truncated. For example, 5/2 will equal 2 in integer division.The modulus operator **%** also works in C/C++ as it does in other languages, yielding the remainder of an integer division. However, you cannot use it on floating-point types.

The following code fragment illustrates **%**:

```
int x, y;
x = 5;
y = 2;
printf("%d ", x/y); /* will display 2 */
printf("%d ", x%y); /* will display 1, the remainder of
the integer division */
```

```
x = 1;
y = 2;
printf("%d %d", x/y, x%y); /* will display 0 1 */
```

The last line prints a 0 and a 1 because 1/2 in integer division is 0 with a remainder of 1.

The unary minus multiplies its operand by –1. That is, any number preceded by a minus sign switches its sign.

**Increment and Decrement**

C/C++ includes two useful operators not found in some other computer languages. These are the increment and decrement operators, ++ and −−. The operator ++ adds 1 to its operand, and −−subtracts 1.

In other words:

```
x = x+1;
```

is the same as

```
++x;
```

and

```
x = x-1;
```

is the same as

```
x--;
```

Both the increment and decrement operators may either precede (prefix) or follow (postfix) the operand.

For example, x = x+1; can be written

```
++x;
```

or

```
x++;
```

There is, however, a difference between the prefix and postfix forms when you use these operators in an expression. When an increment or decrement operator precedes its operand, the increment or decrement operation is performed before obtaining the value of the operand for use in the expression. If the operator follows its operand, the value of the operand is obtained before incrementing or decrementing it.

For instance,

```
x = 10;
```

y = ++x;

sets **y** to 11. However, if you write the code as

x = 10;

y = x++;

**y** is set to 10. Either way, **x** is set to 11; the difference is in when it happens. Most C/C++ compilers produce very fast, efficient object code for increment and decrement operations—code that is better than that generated by using the equivalent assignment statement. For this reason, you should use the increment and decrement operators when you can.

Here is the precedence of the arithmetic operators:

**highest** ++ – –

– (unary minus)

* / %

**lowest** + –

Operators on the same level of precedence are evaluated by the compiler from left to right. Of course, you can use parentheses to alter the order of evaluation. C/C++ treats parentheses in the same way as virtually all other computer languages. Parentheses force an operation, or set of operations, to have a higher level of precedence.

**Relational and Logical Operators**

In the term *relational operator*, relational refers to the relationships that values can have with one another. In the term *logical operator*, logical refers to the ways these relationships can be connected. Because the relational and logical operators oftenwork together, they are discussed together here. The idea of true and false underlies the concepts of relational and logical operators.

In C, true is any value other than zero. False is zero. Expressions that use relational or logical operators return 0 for false and 1 for true. C++ fully supports the zero/non-zero concept of true and false. However, it also defines the **bool** data type and the Boolean constants **true** and **false**. In C++, a 0 value is automatically converted into **false,** and a non-zero value is automatically converted into **true**. The reverse also applies: **true** converts to 1 and **false** converts to 0. In C++, the outcome of a relational or logical operation is **true** or **false**. But since this automatically converts into 1 or 0, the distinction between C and C++ on this issue is mostly academic.

The following program contains the function **xor( )**, which returns the outcome of an exclusive OR operation performed on its two arguments:

```
#include <stdio.h>
int xor(int a, int b);
int main(void)
{
printf("%d", xor(1, 0));
printf("%d", xor(1, 1));
printf("%d", xor(0, 1));
printf("%d", xor(0, 0));
return 0;
}
```

**Bitwise Operators**

Unlike many other languages, C/C++ supports a full complement of bitwise operators.Since C was designed to take the place of assembly language for most programming including operations on bits. *Bitwise operation* refers to testing, setting, or shifting the actual bits in a byte or word, which correspond to the **char** and **int** data types and variants. You cannot use bitwise operations on **float**, **double**, **long double**, **void**, **bool**, or other, more complex types. These operations are applied to the individual bits of the operands. The bitwise AND, OR, and NOT (one's complement) are governed by the same truth table as their logical equivalents, except that they work bit by bit

**The ? Operator**

C/C++ contains a very powerful and convenient operator that replaces certain statements of the if-then-else form. The ternary operator **?** takes the general form *Exp1 ? Exp2 : Exp3;* where *Exp1*, *Exp2*, and *Exp3* are expressions. Notice the use and placement of the colon. The **?** operator works like this: *Exp1* is evaluated. If it is true, *Exp2* is evaluated and becomes the value of the expression. If *Exp1* is false, *Exp3* is evaluated and its value becomes the value of the expression.

For example, in

```
x = 10;
y = x>9 ? 100 : 200;
```

**y** is assigned the value 100. If **x** had been less than 9, **y** would have received the value 200. The same code written using the **if-else** statement is

```
x = 10;
if(x>9) y = 100;
else y = 200;
```

The **?** operator will be discussed more fully in Chapter 3 in relationship to the other conditional statements.

**The & and * Pointer Operators**

A *pointer* is the memory address of some object. A *pointer variable* is a variable that is specifically declared to hold a pointer to an object of its specified type. Knowing a variable's address can be of great help in certain types of routines. However, pointers have three main functions in C/C++. They can provide a fast means of referencing array elements. They allow functions to modify their calling parameters. Lastly, they support linked lists and other dynamic data structures. Chapter 5 is devoted exclusively to pointers. However, this chapter briefly covers the two operators that are used to manipulate pointers. The first pointer operator is **&**, a unary operator that returns the memory address of its operand. (Remember, a unary operator only requires one operand.)

For example,

```
m = &count;
```

places into **m** the memory address of the variable **count**. This address is the computer's internal location of the variable. It has nothing to do with the value of **count**. You can think of **&** as meaning "the address of." Therefore, the preceding assignment statement means "**m** receives the address of **count**."

**The Compile-Time Operator sizeof**

**sizeof** is a unary compile-time operator that returns the length, in bytes, of the variable or parenthesized type-specifier that it precedes. For example, assuming that integers are 4 bytes and **double**s are 8 bytes,

```
double f;
printf("%d ", sizeof f);
printf("%d", sizeof(int));
```

will display **8 4**.

Remember, to compute the size of a type, you must enclose the typenamein parentheses.

This is not necessary for variable names, although there is no harm done if you do so. C/C++ defines (using **typedef**) a special type called **size_t**, which corresponds loosely to an unsigned integer. Technically, the value returned by **sizeof** is of type **size_t**. For all practical purposes, however, you can think of it (and use it) as if it were an unsigned integer value.

**The Comma Operator**

The comma operator strings together several expressions. The left side of the comma operator is always evaluated as **void**. This means that the expression on the right side becomes the value of the total comma-separated expression. For example, x = (y=3, y+1); first assigns **y** the value 3 and then assigns **x** the value 4. The parentheses are necessary because the comma operator has a lower precedence than the assignment operator. Essentially, the comma causes a sequence of operations. When you use it on the right side of an assignment statement, the value assigned is the value of the last expression of the comma-separated list. The comma operator has somewhat the same meaning as the word "and" in normal English as used in the phrase "do this and this and this."

**The Dot (.) and Arrow ( >) Operators**

In C, the **.** (dot) and the >(arrow) operators access individual elements of structures and unions. *Structures* and *unions* are compound (also called *aggregate*) data types that may be referenced under a single name (see Chapter 7). In C++, the dot and arrow operators are also used to access the members of a class. The dot operator is used when working with a structure or union directly. The arrow operator is used when a pointer to a structure or union is used.

For example, given the fragment

```
struct employee
{
char name[80];
int age;
float wage;
} emp;
```

struct employee *p = &emp; /* address of emp into p */

you would write the following code to assign the value 123.23 to the **wage** member of structure variable **emp**:

emp.wage = 123.23;

However, the same assignment using a pointer to **emp** would be p->wage = 123.23.

**The [ ] and ( ) Operators**

Parentheses are operators that increase the precedence of the operations inside them. Square brackets perform array indexing (arrays are discussed fully in Chapter 4). Given an array, the expression within square brackets provides an index into that array.

For example,

```
#include <stdio.h>
char s[80];
int main(void)
{
s[3] = 'X';
printf("%c", s[3]);
return 0;
}
```

first assigns the value '**X**' to the fourth element (remember, all arrays begin at 0) of array **s**, and then prints that element.

**Expressions**

Operators, constants, and variables are the constituents of expressions. An *expression* in C/C++ is any valid combination of these elements. Because most expressions tend to follow the general rules of algebra, they are often taken for granted. However, a few aspects of expressions relate specifically to C and C++.

**Order of Evaluation**

Neither C nor C++ specifies the order in which the subexpressions of an expression are evaluated. This leaves the compiler free to rearrange an expression to produce more optimal code. However, it also means that your code should never rely upon the order in which subexpressions are evaluated. For example, the expression

x = f1() + f2();

does not ensure that **f1( )** will be called before **f2( )**.

**Type Conversion in Expressions**

When constants and variables of different types are mixed in an expression, they are all converted to the same type. The compiler converts all operands up to the type of the largest operand, which is called *type promotion*. First, all **char** and **short int** values are automatically elevated to **int**. (This process is called *integral promotion*.) Once this step has been completed, all other conversions are done operation by operation, as described in the following type conversion algorithm:

IF an operand is a **long double**
THEN the second is converted to **long double**
ELSE IF an operand is a **double**
THEN the second is converted to **double**
ELSE IF an operand is a **float**
THEN the second is converted to **float**
ELSE IF an operand is an **unsigned long**
THEN the second is converted to **unsigned long**
ELSE IF an operand is **long**
THEN the second is converted to **long**
ELSE IF an operand is **unsigned int**
THEN the second is converted to **unsigned int**

There is one additional special case: If one operand is **long** and the other is **unsigned int**, and if the value of the **unsigned int** cannot be represented by a **long**, both operands are converted to **unsigned long**. Once these conversion rules have been applied, each pair of operands is of the same type and the result of each operation is the same as the type of both operands.

For example, consider the type conversions that occur in Figure 2-2. First, the character **ch** is converted to an integer. Then the outcome of **ch/i** is converted to a **double** because **f*d** is **double**. The outcome of **f+i** is **float**, because **f** is a **float**. The final result is **double**.

**Casts**

You can force an expression to be of a specific type by using a *cast*. The general form of a cast is

*(type) expression*

where *type* is a valid data type. For example, to make sure that the expression **x/2** evaluates to type **float**, write (float) x/2 Casts are technically operators. As an operator, a cast is unary and has the same precedence as any other unary operator.

**Spacing and Parentheses**

You can add tabs and spaces to expressions to make them easier to read. For example, the following two expressions are the same:

x=10/y~(127/x);

x = 10 / y ~(127/x);

**Compound Assignments**

There is a variation on the assignment statement, called *compound assignment*, that simplifies the coding of a certain type of assignment operation. For example,

x = x+10;

can be written as

x += 10;

The operator **+=** tells the compiler to assign to **x** the value of **x** plus 10. Compound assignment operators exist for all the binary operators (those that require two operands). In general, statements like:

*var = var operator expression*

can be rewritten as

*var operator = expression*

For another example,

x = x-100;

is the same as

x -= 100;

Compound assignment is widely used in professionally written C/C++ programs; you should become familiar with it. Compound assignment is also commonly referred to as *shorthand assignment* because it is more compact.

**Selection Statements**

C/C++ supports two types of selection statements: **if** and **switch**. In addition, the **?** operator is an alternative to **if** in certain circumstances.

**if**

The general form of the **if** statement is

if (*expression*) *statement*;

else *statement*;

where a *statement* may consist of a single statement, a block of statements, or nothing (in the case of empty statements). The **else** clause is optional.

**Nested ifs**

A nested **if** is an **if** that is the target of another **if** or **else**. Nested **if**s are very common in programming. In a nested **if**, an **else** statement always refers to the nearest **if** statement that is within the same block as the **else** and that is not already associated with an **else**.

For example if(i)

{

if(j) statement 1;

if(k) statement 2; /* this if */

else statement 3; /* is associated with this else */

}

else statement 4; /* associated with if(i) */

As noted, the final **else** is not associated with **if(j)** because it is not in the same block. Rather, the final **else** is associated with **if(i)**. Also, the inner **else** is associated with **if(k)**, which is the nearest **if** .

**The if-else-if Ladder**

A common programming construct is the *if-else-if ladder*, sometimes called the *if-else-if staircase* because of its appearance.

Its general form is

if (*expression*) *statement*;
else
if (*expression*) *statement*;
else
if *(expression*) *statement;*
….
else *statement*;

The conditions are evaluated from the top downward. As soon as a true condition is found, the statement associated with it is executed and the rest of the ladder is bypassed. If none of the conditions are true, the final **else** is executed. That is, if all other conditional tests fail, the last **else** statement is performed. If the final **else** is not present, no action takes place if all other conditions are false. Although the indentation of the preceding if-else-if ladder is technically correct, it can lead to overly deep indentation. For this reason, the if-else-if ladder is generally indented like this:

if (*expression*)
*statement*;
else if (*expression*)
*statement*;
else if (*expression*)
*statement*;
... else
*statement*;

**The ? Alternative**

You can use the **?** operator to replace **if-else** statements of the general form:

if(*condition*) *expression*;
else *expression*;

However, the target of both **if** and **else** must be a single expression—not another statement. The **?** is called a *ternary operator* because it requires three operands.

It takes the general form

*Exp1 ? Exp2 : Exp3*

where *Exp1*, *Exp2*, and *Exp3* are expressions. Notice the use and placement of the colon.

**The Conditional Expression**

Sometimes newcomers to C/C++ are confused by the fact that you can use any valid expression to control the **if** or the **?** operator. That is, you are not restricted to expressions involving the relational and logical operators (as is the case in languages like BASIC or Pascal). The expression must simply evaluate to either a true or false (zero or nonzero) value. For example, the following program reads two integers from the keyboard and displays the quotient. It uses an **if** statement, controlled by the second number, to avoid a divide-by-zero error.

```
/* Divide the first number by the second. */
#include <stdio.h>
int main(void)
{
int a, b;
printf("Enter two numbers: ");
scanf("%d%d", &a, &b);
if(b) printf("%d\n", a/b);
else printf("Cannot divide by zero.\n");
return 0;
}
```

**switch**

C/C++ has a built-in multiple-branch selection statement, called **switch**, which successively tests the value of an expression against a list of integer or character constants. When a match is found, the statements associated with that constant are executed.

The general form of the **switch** statement is

```
switch (expression) {
case constant1:
statement sequence
break;
case constant2:
statement sequence
```

break;

case *constant*3:

*statement sequence*

break;

...

default

*statement sequence*

}

The *expression* must evaluate to a character or integer value. Floating-point expressions, for example, are not allowed. The value of *expression* is tested, in order, against the values of the constants specified in the **case** statements. When a match is found, the statement sequence associated with that **case** is executed until the **break** statement or the end of the **switch** statement is reached. The **default** statement is executed if no matches are found. The **default** is optional and, if it is not present, no action takes place if all matches fail.

**Nested switch Statements**

You can have a **switch** as part of the statement sequence of an outer **switch**. Even if the **case** constants of the inner and outer **switch** contain common values, no conflicts arise.

For example, the following code fragment is perfectly acceptable:

```
switch(x)
{
case 1:
switch(y) {
case 0: printf("Divide by zero error.\n");
break;
case 1: process(x,y);
}
break;
case 2:
.
.
.
```

**Iteration Statements**

In C/C++, and all other modern programming languages, iteration statements (also called *loops*) allow a set of instructions to be executed repeatedly until a certain condition is reached. This condition may be predefined (as in the **for** loop), or open-ended (as in the **while** and **do-while** loops).

**The for Loop**

The general design of the **for** loop is reflected in some form or another in all procedural programming languages. However, in C/C++, it provides unexpected flexibility and power.

The general form of the **for** statement is for(*initialization*; *condition*; *increment*) *statement*;

**for Loop Variations**

The previous discussion described the most common form of the **for** loop. However, several variations of the **for** are allowed that increase its power, flexibility, and applicability to certain programming situations. One of the most common variations uses the comma operator to allow two or more variables to control the loop. (Remember, you use the comma operator to string together a number of expressions in a "do this and this" fashion.)

For example, the variables **x** and **y** control the following loop, and both are initialized inside the **for** statement:

```
for(x=0, y=0; x+y<10; ++x) {
y = getchar();
y = y - '0'; /* subtract the ASCII code for 0
from y */
.
.
.
}
```

**The Infinite Loop**

Although you can use any loop statement to create an infinite loop, **for** is traditionally used for this purpose. Since none of the three expressions that form the **for** loop are required, you can make an endless loop by leaving the conditional expression empty:

```
for( ; ; )
printf("This loop will run forever.\n");
```

When the conditional expression is absent, it is assumed to be true. You may have an initialization and increment expression, but C++ programmers more commonly use the **for(;;)** construct to signify an infinite loop.

**The while Loop**

The second loop available in C/C++ is the **while** loop.

Its general form is

while(*condition*) *statement*;

where *statement* is either an empty statement, a single statement, or a block of statements. The *condition* may be any expression, and true is any nonzero value. The loop iterates while the condition is true. When the condition becomes false, program control passes to the line of code immediately following the loop.

**The do-while Loop**

Unlike **for** and **while** loops, which test the loop condition at the top of the loop, the **do-while** loop checks its condition at the bottom of the loop. This means that a **do-while** loop always executes at least once.

The general form of the **do-while** loop is

do {

*statement*;

} while(*condition*);

**Jump Statements**

C/C++ has four statements that perform an unconditional branch: **return**, **goto**, **break**, and **continue**. Of these, you may use **return** and **goto** anywhere in your program. You may use the **break** and **continue** statements in conjunction with any of the loop statements. As discussed earlier in this chapter, you can also use **break** with **switch**.

**The return Statement**

The **return** statement is used to return from a function. It is categorized as a jump statement because it causes execution to return (jump back) to the point at which the call to the function was made. A **return** may or may not have a value associated with it. If **return** has a value associated with it, that value becomes the return value of the function. In C89, a non-**void** function does not technically have to return a value. If no return value is specified, a garbage value is returned. However, in C++ (and in C99), a non-**void** function *must* return a value. That

is, in C++, if a function is specified as returning a value, any **return** statement within it must have a value associated with it. (Even in C89, if a function is declared as returning a value, it is good practice to actually return one!)

The general form of the **return** statement is

return *expression*;

The *expression* is present only if the function is declared as returning a value. In this case, the value of *expression* will become the return value of the function.

**The goto Statement**

Since C/C++ has a rich set of control structures and allows additional control using **break** and **continue**, there is little need for **goto**. Most programmers' chief concern about the **goto** is its tendency to render programs unreadable. Nevertheless, although the **goto** statement fell out of favor some years ago, it occasionally has its uses. There are no programming situations that require **goto**. Rather, it is a convenience, which, if used wisely, can be a benefit in a narrow set of programming situations, such as jumping out of a set of deeply nested loops. The **goto** is not used outside of this section. The **goto** statement requires a label for operation. (A *label* is a valid identifier followed by a colon.) Furthermore, the label must be in the same function as the **goto** that uses it—you cannot jump between functions.

The general form of the **goto**

statement is

goto *label*;

...

*label*:

where *label* is any valid label either before or after **goto**. For example, you could create a loop from 1 to 100 using the **goto** and a label, as shown here:

```
x = 1;
loop1:
x++;
if(x<100) goto loop1;
```

**The break Statement**

The **break** statement has two uses. You can use it to terminate a **case** in the **switch** statement (covered in the section on **switch** earlier in this chapter). You can also use it to force

immediate termination of a loop, bypassing the normal loop conditional test. When the **break** statement is encountered inside a loop, the loop is immediately terminated and program control resumes at the next statement following the loop.

For example,

```
#include <stdio.h>
int main(void)
{
int t;
for(t=0; t<100; t++) {
printf("%d ", t);
if(t==10) break;
}
return 0;
}
```

prints the numbers 0 through 10 on the screen. Then the loop terminates because **break** causes immediate exit from the loop, overriding the conditional test **t<100**.

**The exit( ) Function**

Although **exit( )** is not a program control statement, a short digression that discusses it is in order at this time. Just as you can break out of a loop, you can break out of a program by using the standard library function **exit( )**. This function causes immediate termination of the entire program, forcing a return to the operating system. In effect, the **exit( )** function acts as if it were breaking out of the entire program.

The general form of the **exit( )** function is

void exit(int *return_code*);

**The continue Statement**

The **continue** statement works somewhat like the **break** statement. Instead of forcing termination, however, **continue** forces the next iteration of the loop to take place, skipping any code in between. For the **for** loop, **continue** causes the conditional test and increment portions of the loop to execute. For the **while** and **do-while** loops, program control passes to the conditional tests. For example, the following program counts the number of spaces contained in the string entered by the user:

```
/* Count spaces */
#include <stdio.h>
int main(void)
{
char s[80], *str;
int space;
printf("Enter a string: ");
gets(s);
str = s;
for(space=0; *str; str++) {
if(*str != ' ') continue;
space++;
}
printf("%d spaces\n", space);
return 0;
}
```

Each character is tested to see if it is a space. If it is not, the **continue** statement forces the **for** to iterate again. If the character *is* a space, **space** is incremented.

**Expression Statements**

Chapter 2 covered expressions thoroughly. However, a few special points are mentioned here. Remember, an expression statement is simply a valid expression followed by a semicolon, as in

```
func(); /* a function call */
a = b+c; /* an assignment statement */
b+f(); /* a valid, but strange statement */
; /* an empty statement */
```

The first expression statement executes a function call. The second is an assignment. The third expression, though strange, is still evaluated by the C++ compiler and the function **f( )** is called. The final example shows that a statement can be empty (sometimes called a *null statement*).

**Block Statements**

Block statements are simply groups of related statements that are treated as a unit. The statements that make up a block are logically bound together. Block statements are also called *compound statements*. A block is begun with a **{** and terminated by its matching **}**. Programmers use block statements most commonly to create a multistatement target for some other statement, such as **if**. However, you may place a block statement anywhere you would put any other statement.

For example, this is perfectly valid (although unusual) C/C++ code:

```
#include <stdio.h>
int main(void)
{
int i;
{ /* a block statement */
i = 120;
printf("%d", i);
}
return 0;
```

## 1.5 Arrays and strings

**Single-Dimension Arrays**

The general form for declaring a single-dimension array is

*type var_name[size]*;

Like other variables, arrays must be explicitly declared so that the compiler may allocate space for them in memory. Here, *type* declares the base type of the array, which is the type of each element in the array, and *size* defines how many elements the array will hold. For example, to declare a 100-element array called **balance** of type **double**,

use this statement:

```
double balance[100];
```

An element is accessed by indexing the array name. This is done by placing the index of the element within square brackets after the name of the array.

For example,

balance[3] = 12.23;

assigns element number 3 in **balance** the value 12.23.

**Two-Dimensional Arrays**

C/C++ supports multidimensional arrays. The simplest form of the multidimensional array is the two-dimensional array. A two-dimensional array is, essentially, an array of one-dimensional arrays.

To declare a two-dimensional integer array **d** of size 10,20, you would write int d[10][20];

Pay careful attention to the declaration. Some other computer languages use commas to separate the array dimensions; C/C++, in contrast, places each dimension in its own set of brackets.Similarly, to access point 1,2 of array **d**, you would use d[1][2]

The following example loads a two-dimensional array with the numbers 1 through 12 and prints them row by row.

```
#include <stdio.h>
int main(void)
{
int t, i, num[3][4];
for(t=0; t<3; ++t)
for(i=0; i<4; ++i)
num[t][i] = (t*4)+i+1;
/* now print them out */
for(t=0; t<3; ++t) {
for(i=0; i<4; ++i)
printf("%3d ", num[t][i]);
printf("\n");
}
return 0;
}
```

In this example, **num[0][0]** has the value 1, **num[0][1]** the value 2, **num[0][2]** the value 3, and so on. The value of **num[2][3]** will be 12.

**Arrays of Strings**

It is not uncommon in programming to use an array of strings. For example, the input processor to a database may verify user commands against an array of valid commands. To create an array of null-terminated strings, use a two-dimensional character array. The size of the left index determines the number of strings and the size of the right index specifies the maximum length of each string. The following code declares an array of 30 strings, each with a maximum length of 79 characters, plus the null terminator. char str_array[30][80]; It is easy to access an individual string: You simply specify only the left index.

For example, the following statement calls **gets( )** with the third string in **str_array**.

```
gets(str_array[2]);
```

The preceding statement is functionally equivalent to

```
gets(&str_array[2][0]);
```

but the first of the two forms is much more common in professionally writtenC/C++ code. To better understand how string arrays work, study the following short program, which uses a string array as the basis for a very simple text editor:

```
/* A very simple text editor. */
#include <stdio.h>
#define MAX 100
#define LEN 80
char text[MAX][LEN];
int main(void)
{
register int t, i, j;
printf("Enter an empty line to quit.\n");
for(t=0; t<MAX; t++) {
printf("%d: ", t);
gets(text[t]);
if(!*text[t]) break; /* quit on blank line */
}
for(i=0; i<t; i++) {
for(j=0; text[i][j]; j++) putchar(text[i][j]);
```

```
putchar('\n');
}
return 0;
}
```

This program inputs lines of text until a blank line is entered. Then it redisplays each line one character at a time.

**Multidimensional Arrays**

C/C++ allows arrays of more than two dimensions. The exact limit, if any, is determined by your compiler. The general form of a multidimensional array declaration is

*type name*[*Size1*][*Size2*][*Size3*]. . .[*SizeN*];

Arrays of more than three dimensions are not often used because of the amount of memory they require. For example, a four-dimensional character array with dimensions 10,6,9,4 requires 10 * 6 * 9 * 4 or 2,160 bytes. If the array held 2-byte integers, 4,320 bytes would be needed. If the array held **double**s (assuming 8 bytes per **double**), 17,280 bytes would be required. The storage required increases exponentially with the number of dimensions. For example, if a fifth dimension of size 10 was added to the preceding array, then 172, 800 bytes would be required.

In multidimensional arrays, it takes the computer time to compute each index. This means that accessing an element in a multidimensional array can be slower than accessing an element in a single-dimension array. When passing multidimensional arrays into functions, you must declare all but the leftmost dimension.

For example, if you declare array **m** as int m[4][3][6][5]; a function, **func1( )**, that receives **m**, would look like this:

```
void func1(int d[][3][6][5])
{
.
.
.
}
```

Of course, you can include the first dimension if you like.

## 1.6 Pointers & user-defined types

A *pointer* is a variable that holds a memory address. This address is the location of another object (typically another variable) in memory. For example, if one variable contains the address of another variable, the first variable is said to *point to* the second.

**Pointer Variables**

If a variable is going to hold a pointer, it must be declared as such. Apointer declaration consists of a base type, an *, and the variable name. The general form for declaring a pointer variable is

*type *name;*

where *type* is the base type of the pointer and may be any valid type. The name of the pointer variable is specified by *name.*

The base type of the pointer defines what type of variables the pointer can point to. Technically, any type of pointer can point anywhere in memory. However, all pointer arithmetic is done relative to its base type, so it is important to declare the pointer correctly. (Pointer arithmetic is discussed later in this chapter.)

**The Pointer Operators**

The pointer operators were discussed in Chapter 2. We will take a closer look at them here, beginning with a review of their basic operation. There are two special pointer operators: * and **&**. The **&** is a unary operator that returns the memory address of its operand. (Remember, a unary operator only requires one operand.)

For example,

m = &count;

places into **m** the memory address of the variable **count**. This address is the computer's internal location of the variable. It has nothing to do with the value of **count**. You can think of **&** as returning "the address of." Therefore, the preceding assignment statement means "**m** receives the address of **count**." To understand the above assignment better, assume that the variable **count** uses memory location 2000 to store its value. Also assume that **count** has a value of 100. Then, after the preceding assignment, **m** will have the value 2000. The second pointer operator, *, is the complement of **&**. It is a unary operator that returns the value located at the address that follows.

For example, if **m** contains the memory address of the variable **count**,

```
q = *m;
```

places the value of **count** into **q**. Thus, **q** will have the value 100 because 100 is stored at location 2000, which is the memory address that was stored in **m**. You can think of * as "at address." In this case, the preceding statement means "**q** receives the value at address **m**." Both **&** and * have a higher precedence than all other arithmetic operators except the unary minus, with which they are equal.

You must make sure that your pointer variables always point to the correct type of data. For example, when you declare a pointer to be of type **int**, the compiler assumes that any address that it holds points to an integer variable—whether it actually does or not. Because you can assign any address you want to a pointer variable, the following program compiles without error, but does not produce the desired result:

```
#include <stdio.h>
int main(void)
{
double x = 100.1, y;
int *p;
/* The next statement causes p (which is an integer pointer) to point to a double. */
p = (int *)&x;
/* The next statement does not operate as expected. */
y = *p;
printf("%f", y); /* won't output 100.1 */
return 0;
}
```

This will not assign the value of **x** to **y**. Because **p** is declared as an integer pointer, only 4 bytes of information (assuming 4-byte integers) will be transferred to **y**, not the 8 bytes that normally make up a **double**. *In C++, it is illegal to convert one type of pointer into another without the use of an explicit type cast. In C, casts should be used for most pointer conversions.*

**Pointer Expressions**

In general, expressions involving pointers conform to the same rules as other expressions. This section examines a few special aspects of pointer expressions. Pointer **Assignments** As with

any variable, you may use a pointer on the right-hand side of an assignment statement to assign its value to another pointer.

For example,

```
#include <stdio.h>
int main(void)
{
int x;
int *p1, *p2;
p1 = &x;
p2 = p1;
printf(" %p", p2); /* print the address of x, not x's value! */
return 0;
}
```

Both **p1** and **p2** now point to **x**. The address of **x** is displayed by using the **%p printf( )** format specifier, which causes **printf( )** to display an address in the format used by the host computer.

**Pointer Arithmetic**

There are only two arithmetic operations that you may use on pointers: addition and subtraction. To understand what occurs in pointer arithmetic, let **p1** be an integer pointer with a current value of 2000. Also, assume integers are 2 bytes long. After the expression p1++; **p1** contains 2002, not 2001. The reason for this is that each time **p1** is incremented; it will point to the next integer. The same is true of decrements. For example, assuming that **p1** has the value 2000, the expression p1--; causes **p1** to have the value 1998. Generalizing from the preceding example, the following rules govern pointer arithmetic. Each time a pointer is incremented, it points to the memory location

**Pointer Comparisons**

You can compare two pointers in a relational expression. For instance, given two pointers **p** and **q**, the following statement is perfectly valid: if(p<q) printf("p points to lower memory than q\n");

**Pointers to Functions**

A particularly confusing yet powerful feature of C++ is the *function pointer*. Even though a function is not a variable, it still has a physical location in memory that can be assigned to a

pointer. This address is the entry point of the function and it is the address used when the function is called. Once a pointer points to a function, the function can be called through that pointer. Function pointers also allow functions to be passed as arguments to other functions.

## 1.7 Function Components

**The General Form of a Function**

The general form of a function is

*ret-type function-name*(*parameter list*)

{

*body of the function*

}

The *ret-type* specifies the type of data that the function returns.Afunction may return any type of data except an array. The *parameter list* is a comma-separated list of variable names and their associated types that receive the values of the arguments when the function is called.Afunction may be without parameters, in which case the parameter list is empty. However, even if there are no parameters, the parentheses are still required. In variable declarations, you can declare many variables to be of a common type by using a comma-separated list of variable names. In contrast, all function parameters must be declared individually, each including both the type and name. That is, the parameter declaration list for a function takes this general form:

*f(type varname1, type varname2, . . . , type varnameN)*

For example, here are correct and incorrect function parameter declarations:

```
f(int i, int k, int j) /* correct */
f(int i, k, float j) /* incorrect */
```

**Scope Rules of Functions**

The *scope rules* of a language are the rules that govern whether a piece of code knows about or has access to another piece of code or data. Each function is a discrete block of code. Afunction's code is private to that function and cannot be accessed by any statement in any other function except through a call to that function. (For instance, you cannot use **goto** to jump into the middle of another function.) The code that constitutes the body of a function is hidden from the rest of the program and, unless it uses global variables or data, it can neither

affect nor be affected by other parts of the program. Stated another way, the code and data that are defined within one function cannot interact with the code or data defined in another function because the two functions have a different scope. Variables that are defined within a function are called *local* variables. A local variable comes into existence when the function is entered and is destroyed upon exit. That is, local variables cannot hold their value between function calls. The only exception to this rule is when the variable is declared with the **static** storage class specifier. This causes the compiler to treat the variable as if it were a global variable for storage purposes, but limits its scope to within the function. In C (and C++) you cannot define a function within a function. This is why neither C nor C++ are technically block-structured languages.

### 1.8 Argument passing

**Function Arguments**

If a function is to use arguments, it must declare variables that accept the values of the arguments. These variables are called the *formal parameters* of the function. They behave like other local variables inside the function and are created upon entry into the function and destroyed upon exit. As shown in the following function, the parameter declarations occur after the function name:

```
/* Return 1 if c is part of string s; 0 otherwise. */
int is_in(char *s, char c)
{
while(*s)
if(*s==c) return 1;
else s++;
return 0;
}
```

The function **is_in( )** has two parameters: **s** and **c**. This function returns 1 if the character **c** is part of the string **s**; otherwise, it returns 0. As with local variables, you may make assignments to a function's formal parameters or use them in an expression. Even though these variables perform the special task of receiving the value of the arguments passed to the function, you can use them as you do any other local variable.

**Call by Value, Call by Reference**

In a computer language, there are two ways that arguments can be passed to a subroutine. The first is known as *call by value*. This method copies the *value of* an argument into the formal parameter of the subroutine. In this case, changes made to the parameter have no effect on the argument. *Call by reference* is the second way of passing arguments to a subroutine. In this method, the *address* of an argument is copied into the parameter. Inside the subroutine, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the argument. By default, C/C++ uses call by value to pass arguments. In general, this means that code within a function cannot alter the arguments used to call the function.

Consider the following program:

```
#include <stdio.h>
int sqr(int x);
int main(void)
{
int t=10;
printf("%d %d", sqr(t), t);
return 0;
}
int sqr(int x)
{
x = x*x;
return(x);
}
```

In this example, the value of the argument to **sqr( )**, 10, is copied into the parameter **x**. When the assignment **x = x*x** takes place, only the local variable **x** is modified. The variable **t**, used to call **sqr**( ), still has the value 10. Hence, the output is **100 10**. Remember that it is a copy of the value of the argument that is passed into the function. What occurs inside the function has no effect on the variable used in the call.

**Creating a Call by Reference**

Even though C/C++ uses call by value for passing parameters, you can create a call by reference by passing a pointer to an argument, instead of the argument itself. Since the address of the argument is passed to the function, code within the function can change the value of the argument outside the function. Pointers are passed to functions just like any other value. Of course, you need to declare the parameters as pointer types.

For example, the function **swap( )**, which exchanges the values of the two integer variables pointed to by its arguments, shows how.

```
void swap(int *x, int *y)
{
int temp;
temp = *x; /* save the value at address x */
*x = *y; /* put y into x */
*y = temp; /* put x into y */
}
```

**swap( )** is able to exchange the values of the two variables pointed to by **x** and **y** because their addresses (not their values) are passed. Thus, within the function, the contents of the variables can be accessed using standard pointer operations, and the contents of the variables used to call the function are swapped.

**The return Statement**

The **return** statement itself is described in Chapter 3. As explained, it has two important uses. First, it causes an immediate exit from the function that it is in. That is, it causes program execution to return to the calling code. Second, it may be used to return a value.

This section examines how the **return** statement is used.

## 1.9 Inline functions

**Inline Functions**

There is an important feature in C++, called an *inline function*, that is commonly used with classes. Since the rest of this chapter (and the rest of the book) will make heavy use of it, inline functions are examined here. In C++, you can create short functions that are not actually called; rather, their code is expanded in line at the point of each invocation. This process is

similar to using a function-like macro. To cause a function to be expanded in line rather than called, precede its definition with the **inline** keyword.

For example, in this program, the function **max( )** is expanded in line instead of called:

```
#include <iostream>
using namespace std;
inline int max(int a, int b)
{
return a>b ? a : b;
}
int main()
{
cout << max(10, 20);
cout << " " << max(99, 88);
return 0;
}
```

As far as the compiler is concerned, the preceding program is equivalent to this one:

```
#include   <iostream>
using  namespace  std;
int main()
{
cout << (10>20 ? 10 : 20);
cout << " " << (99>88 ? 99 : 88);
return 0;
}
```

The reason that **inline** functions are an important addition to C++ is that they allow you to create very efficient code. Since classes typically require several frequently executed interface functions (which provide access to private data), the efficiency of these functions is of critical concern. As you probably know, each time a function is called, a significant amount of overhead is generated by the calling and return mechanism. Typically, arguments are pushed onto the stack and various registers are saved when a function is called, and then restored when the function returns. The trouble is that these instructions take time. However, when a function

is expanded in line, none of those operations occur. Although expanding function calls in line can produce faster run times, it can also result in larger code size because of duplicated code. For this reason, it is best to **inline** only very small functions. Further, it is also a good idea to **inline** only those functions that will have significant impact on the performance of your program.

**Defining Inline Functions Within a Class**

It is possible to define short functions completely within a class declaration. When a function is defined inside a class declaration, it is automatically made into an **inline** function (if possible). It is not necessary (but not an error) to precede its declaration with the **inline** keyword. For example, the preceding program is rewritten here with the definitions of **init( )** and **show( )** contained within the declaration of **myclass**:

```
#include <iostream>
using namespace std;
class myclass {
int a, b;
public:
// automatic inline
void init(int i, int j) { a=i; b=j; }
void show() { cout << a << " " << b << "\n"; }
};
int main()
{
myclass x;
x.init(10, 20);
x.show();
return 0;
}
```

Notice the format of the function code within **myclass**.

## 1.10 Function overloading

**Function Overloading**

Function overloading is the process of using the same name for two or more functions. The secret to overloading is that each redefinition of the function must use either different types of parameters or a different number of parameters. It is only through these differences that the compiler knows which function to call in any given situation.

For example, this program overloads **myfunc( )** by using different types of parameters.

```
#include <iostream>
using namespace std;
int myfunc(int i); // these differ in types of parameters
double myfunc(double i);
int main()
{
cout << myfunc(10) << " "; // calls myfunc(int i)
cout << myfunc(5.4); // calls myfunc(double i)
return 0;
}
double myfunc(double i)
{
return i;
}
int myfunc(int i)
{
return i;
}
```

The next program overloads **myfunc( )** using a different number of parameters:

```
#include <iostream>
using namespace std;
int myfunc(int i); // these differ in number of parameters
int myfunc(int i, int j);
int main()
```

```
{
cout << myfunc(10) << " "; // calls myfunc(int i)
cout << myfunc(4, 5); // calls myfunc(int i, int j)
return 0;
}
int myfunc(int i)
{
return i;
}
int myfunc(int i, int j)
{
return i*j;
}
```

As mentioned, the key point about function overloading is that the functions must differ in regard to the types and/or number of parameters. Two functions differing only in their return types cannot be overloaded. For example, this is an invalid attempt to overload **myfunc( )**:

int myfunc(int i); // Error: differing return types are float myfunc(int i); // insufficient when overloading. Sometimes, two function declarations will appear to differ, when in fact they do not.

For example, consider the following declarations.

```
void f(int *p);
void f(int p[]); // error, *p is same as p[]
```

Remember, to the compiler ***p** is the same as **p[ ].** Therefore, although the two prototypes appear to differ in the types of their parameter, in actuality they do not.

## 1.11 Recursive functions

**Recursion**

In C/C++, a function can call itself. A function is said to be *recursive* if a statement in the body of the function calls itself. Recursion is the process of defining something in terms of itself, and is sometimes called *circular definition*.

A simple example of a recursive function is **factr( )**, which computes the factorial of an integer. The factorial of a number **n** is the product of all the whole numbers between 1 and **n**. For example, 3 factorial is 1 x 2 x 3, or 6. Both **factr( )** and its iterative equivalent are shown here:

```
/* recursive */
int factr(int n) {
int answer;
if(n==1) return(1);
answer = factr(n-1)*n; /* recursive call */
return(answer);
}
/* non-recursive */
int fact(int n) { int
t, answer; answer
= 1; for(t=1; t<=n;
t++)
answer=answer*(t);
return(answer);
}
```

The nonrecursive version of **fact( )** should be clear. It uses a loop that runs from 1 to **n** and progressively multiplies each number by the moving product. The operation of the recursive **factr( )** is a little more complex. When **factr( )** is called with an argument of 1, the function returns 1. Otherwise, it returns the product of **factr(n−1)*n**. To evaluate this expression, **factr( )** is called with **n−1**. This happens until **n** equals 1 and the calls to the function begin returning. Computing the factorial of 2, the first call to **factr( )** causes a second, recursive call with the argument of 1. This call returns 1, which is then multiplied by 2 (the original **n** value). The answer is then 2. Try working through the computation of 3 factorial on your own. (You might want to insert **printf( )** statements into **factr( )** to see the level of each call and what the intermediate answers are.) When a function calls itself, a new set of local variables and parameters are allocated storage on the stack, and the function code is executed from the top with these new variables. A recursive call does not make a new copy of the function. Only the

values being operated upon are new. As each recursive call returns, the old local variables and parameters are removed from the stack and execution resumes at the point of the function call inside the function. Recursive functions could be said to "telescope" out and back.

Often, recursive routines do not significantly reduce code size or improve memory utilization over their iterative counterparts. Also, the recursive versions of most routines may execute a bit slower than their iterative equivalents because of the overhead of the repeated function calls. In fact, many recursive calls to a function could cause a stack overrun. Because storage for function parameters and local variables is on the stack and each new call creates a new copy of these variables, the stack could be exhausted. However, you probably will not have to worry about this unless a recursive function runs wild.

The main advantage to recursive functions is that you can use them to create clearer and simpler versions of several algorithms. For example, the Quicksort algorithm is difficult to implement in an iterative way. Also, some problems, especially ones related to artificial intelligence, lend themselves to recursive solutions. Finally, some people seem to think recursively more easily than iteratively. When writing recursive functions, you must have a conditional statement, such as an **if,** somewhere to force the function to return without the recursive call being executed. If you don't, the function will never return once you call it. Omitting the conditional statement is a common error when writing recursive functions. Use **printf( )** liberally during program development so that you can watch what is going on and abort execution if you see a mistake.

# UNIT 2

# Classes & Objects – I

**2.1 Class Specification**

**2.2 Class Objects**

**2.3 Scope resolution operator**

**2.4 Access members**

**2.5 Defining member functions**

**2.6 Data hiding**

**2.7 Constructors**

**2.8 Destructors**

**2.9 Parameterized constructors**

**2.10 Static data members**

**2.11 Functions**

## 2.1 Class Specification

**Classes**

Classes are created using the keyword **class**. A class declaration defines a new type that links code and data. This new type is then used to declare objects of that class. Thus, a class is a logical abstraction, but an object has physical existence. In other words, an object is an *instance* of a class. A class declaration is similar syntactically to a structure. A simplified general form of a class declaration was shown. Here is the entire general form of a **class** declaration that does not inherit any other class.

class *class-name {*
*private data and functions*
*access-specifier:*
*data and functions*
*access-specifier:*
*data and functions*
*// ...*
*access-specifier:*
*data and functions*
*} object-list;*

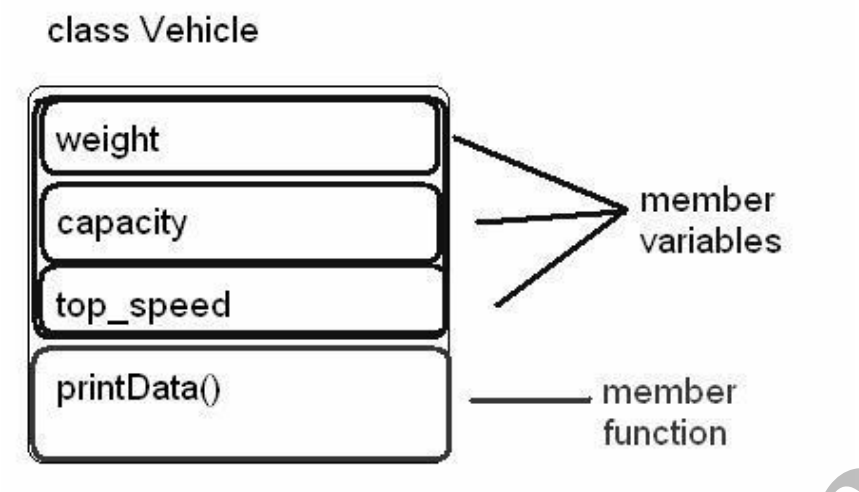
The *object-list* is optional. If present, it declares objects of the class. Here, *access-specifier*is one of these three C++ keywords:

public
private
protected

## 2.2 Class Objects

**Classes and objects**

C++ has classes **A class** is a user-defined type. The variables of this type are **objects.** A class can be obtained from a structure if some member functions are added.




Member data and member functions can both be accessed using variable-to-member access operator Each object will have separate separate copy of the member data within itself but only one copy of member function exists

**Private and public**

A member (variable or function) can be **private** or **public** The keywords are also known as access modifiers or access specifiers a "good" class keeps its member variables private = **data hiding** and uses public member functions to access or change each private variable. Class members are private by default whereas struct members are public Objects Variables of classes are known as objects.

## 2.3 Scope resolution operator

**The Scope Resolution Operator**

As you know, the **::** operator links a class name with a member name in order to tell the compiler what class the member belongs to. However, the scope resolution operator has another related use: it can allow access to a name in an enclosing scope that is "hidden" by a local declaration of the same name.

For example, consider this fragment:

```
int i; // global i
void f()
```

```
{
int i; // local i
i = 10; // uses local i
.
.
.
}
```

As the comment suggests, the assignment **i = 10** refers to the local **i**. But what if function **f( )** needs to access the global version of **i**? It may do so by preceding the **i** with the **::** operator, as shown here.

```
int i; // global i
void f()
{
int i; // local i
::i = 10; // now refers to global i
.
.
.
}
```

## 2.4 Access members

The arrow operator is used to access members of the object. Here is a short program that creates a class called **balance** that links a person's name with his or her account balance. Inside **main( )**, an object of type **balance** is created dynamically.

```
#include <iostream>
#include <new>
#include <cstring>
using namespace std;
class balance {
double cur_bal;
char name[80];
```

```
public:
void set(double n, char *s) {
cur_bal = n;
strcpy(name, s);
}
void get_bal(double &n, char *s) {
n = cur_bal;
strcpy(s, name);
}
};
int main()
{
balance *p;
char s[80];
double n;
try {
p = new balance;
} catch (bad_alloc xa) {
cout << "Allocation Failure\n";
return 1;
}
p->set(12387.87, "Ralph Wilson");
p->get_bal(n, s);
cout << s << "'s balance is: " << n;
cout << "\n";
delete p;
return 0;
}
```

Because **p** contains a pointer to an object, the arrow operator is used to access members of the object.

## 2.5 Defining member functions

A member operator function takes this general form:

*ret-type class-name::*operator#*(arg-list)*

{

// operations

}

Often, operator functions return an object of the class they operate on, but *ret-type* can be any valid type. The # is a placeholder. When you create an operator function, substitute the operator for the #. For example, if you are overloading the / operator, use **operator/.** When you are overloading a unary operator, *arg-list* will be empty. When you are overloading binary operators, *arg-list* will contain one parameter. (The reasons for this seemingly unusual situation will be made clear in a moment.) Here is a simple first example of operator overloading. This program creates a class called **loc**, which stores longitude and latitude values. It overloads the + operator relative to this class. Examine this program carefully, paying special attention to the definition of **operator+()**:

```
#include <iostream>
using namespace std;
class loc {
int longitude, latitude;
public:
loc() {}
loc(int lg, int lt) {
longitude = lg;
latitude = lt;
}
void show() {
cout << longitude << " ";
cout << latitude << "\n";
}
loc operator+(loc op2);
};
```

```
// Overload + for loc.
loc loc::operator+(loc op2)
{
loc temp;
temp.longitude = op2.longitude + longitude;
temp.latitude = op2.latitude + latitude;
return temp;
}
int main()
{
loc ob1(10, 20), ob2( 5, 30);
ob1.show(); // displays 10 20
ob2.show(); // displays 5 30
ob1 = ob1 + ob2;
ob1.show(); // displays 15 50
return 0;
}
```

## 2.6 Data hiding

*Data hiding* is the mechanism that binds together code and the data it manipulates, and keeps both safe from outside interference and misuse. In an object-oriented language, code and data may be combined in such a way that a self-contained "black box" is created. When code and data are linked together in this fashion, an *object* is created. In other words, an object is the device that supports encapsulation. Within an object, code, data, or both may be *private* to that object or *public*. Private code or data is known to and accessible only by another part of the object. That is, private code or data may not be accessed by a piece of the program that exists outside the object. When code or data is public, other parts of your program may access it even though it is defined within an object. Typically, the public parts of an object are used to provide a controlled interface to the private elements of the object. For all intents and purposes, an object is a variable of a user-defined type. It may seem strange that an object that links both

code and data can be thought of as a variable. However, in object-oriented programming, this is precisely the case. Each time you define a new type of object, you are creating a new data type. Each specific instance of this data type is a compound variable.

## 2.7 Constructors

**Constructors**

It is very common for some part of an object to require initialization before it can be used. For example, think back to the **stack** class developed earlier in this chapter. Before the stack could be used, **tos** had to be set to zero. This was performed by using the function **init( ).** Because the requirement for initialization is so common, C++ allows objects to initialize themselves when they are created. This automatic initialization is performed through the use of a constructor function.

A *constructor* is a special function that is a member of a class and has the same name as that class. For example, here is how the **stack** class looks when converted to use a constructor for initialization:

```
// This creates the class stack.
class stack {
int stck[SIZE];
int tos;
public:
stack(); // constructor
void push(int i);
int pop();
};
```

Notice that the constructor **stack( )** has no return type specified. In C++, constructors cannot return values and, thus, have no return type.

The **stack( )** constructor is coded like this:

```
// stack's constructor
stack::stack()
{
```

```
tos = 0;
cout << "Stack Initialized\n";
}
```

Keep in mind that the message **Stack Initialized** is output as a way to illustrate the constructor. In actual practice, most constructors will not output or input anything. They will simply perform various initializations. An object's constructor is automatically called when the object is created. This means that it is called when the object's declaration is executed.

## 2.8 Destructors

**Destructors**

The complement of the constructor is the *destructor*. In many circumstances, an object will need to perform some action or actions when it is destroyed. Local objects are created when their block is entered, and destroyed when the block is left. Global objects are destroyed when the program terminates. When an object is destroyed, its destructor (if it has one) is automatically called. There are many reasons why a destructor may be needed. For example, an object may need to deallocate memory that it had previously allocated or it may need to close a file that it had opened. In C++, it is the destructor that handles deactivation events. The destructor has the same name as the constructor, but it is preceded by a ~. For example, here is the **stack** class and its constructor and destructor. (Keep in mind that the **stack** class does not require a destructor; the one shown here is just for illustration.)

```
// This creates the class stack.
class stack {
int stck[SIZE];
int tos;
public:
stack(); // constructor
~stack(); // destructor
void push(int i);
int pop();
};
// stack's constructor
```

```
stack::stack()
{
tos = 0;
cout << "Stack Initialized\n";
}
// stack's destructor
stack::~stack()
{
cout << "Stack Destroyed\n";
}
```

Notice that, like constructors, destructors do not have return values. To see how constructors and destructors work, here is a new version of the **stack** program examined earlier in this chapter. Observe that **init( )** is no longer needed.

```
// Using a constructor and destructor.
#include <iostream>
using namespace std;
#define SIZE 100
// This creates the class stack.
class stack {
int stck[SIZE];
int tos;
public:
stack(); // constructor
~stack(); // destructor
void push(int i);
int pop();
};
// stack's constructor
stack::stack()
{
tos = 0;
```

```
cout << "Stack Initialized\n";
}
// stack's destructor
stack::~stack()
{
cout << "Stack Destroyed\n";
}
void stack::push(int i)
{
if(tos==SIZE) {
cout << "Stack is full.\n";
return;
}
stck[tos] = i;
tos++;
}
int stack::pop()
{
if(tos==0) {
cout << "Stack underflow.\n";
return 0;
}
tos--;
return stck[tos];
}
int main()
{
stack a, b; // create two stack objects
a.push(1);
b.push(2);
a.push(3);
```

```
b.push(4);
cout << a.pop() << " ";
cout << a.pop() << " ";
cout << b.pop() << " ";
cout << b.pop() << "\n";
return 0;
}
```

This program displays the following:

```
Stack Initialized
Stack Initialized
3 1 4 2
Stack Destroyed
Stack Destroyed
```

## 2.9 Parameterized constructors

**Parameterized Constructors**

It is possible to pass arguments to constructors. Typically, these arguments help initialize an object when it is created. To create a parameterized constructor, simply add parameters to it the way you would to any other function. When you define the constructor's body, use the parameters to initialize the object. For example, here is a simple class that includes a parameterized constructor:

```
#include <iostream>
using namespace std;
class myclass {
int a, b;
public:
myclass(int i, int j) {a=i; b=j;}
void show() {cout << a << " " << b;}
};
int main()
{
```

```
myclass ob(3, 5);
ob.show();
return 0;
}
```

Notice that in the definition of **myclass( ),** the parameters **i** and **j** are used to give initial values to **a** and **b**.

The program illustrates the most common way to specify arguments when you declare an object that uses a parameterized constructor. Specifically, this statement myclass ob(3, 4); causes an object called **ob** to be created and passes the arguments **3** and **4** to the **i** and **j** parameters of **myclass( )**. You may also pass arguments using this type of declaration statement:

```
myclass ob = myclass(3, 4);
```

However, the first method is the one generally used, and this is the approach taken by most of the examples in this book. Actually, there is a small technical difference between the two types of declarations that relates to copy constructors. (Copy constructors are discussed later) Here is another example that uses a parameterized constructor. It creates a class that stores information about library books.

```
#include <iostream>
#include <cstring>
using namespace std;
const int IN = 1;
const int CHECKED_OUT = 0;
class book {
char author[40];
char title[40];
int status;
public:
book(char *n, char *t, int s);
int get_status() {return status;}
void set_status(int s) {status = s;}
void show();
```

```
};
book::book(char *n, char *t, int s)
{
strcpy(author, n);
strcpy(title, t);
status = s;
}
void book::show()
{
cout << title << " by " << author;
cout << " is ";
if(status==IN) cout << "in.\n";
else cout << "out.\n";
}
int main()
{
book b1("Twain", "Tom Sawyer", IN);
book b2("Melville", "Moby Dick", CHECKED_OUT);
b1.show();
b2.show();
return 0;
}
```

Parameterized constructors are very useful because they allow you to avoid having to make an additional function call simply to initialize one or more variables in an object. Each function call you can avoid makes your program more efficient. Also, notice that the short **get_status( )** and **set_status( )** functions are defined in line, within the **book** class. This is a common practice when writing C++ programs.

## 2.10 Static data members

**Static Data Members**

When you precede a member variable's declaration with **static**, you are telling the compiler that only one copy of that variable will exist and that all objects of the class will share that variable. Unlike regular data members, individual copies of a **static** member variable are not made for each object. No matter how many objects of a class are created, only one copy of a **static** data member exists. Thus, all objects of that class use that same variable. All **static** variables are initialized to zero before the first object is created. When you declare a **static** data member within a class, you are *not* defining it. (That is, you are not allocating storage for it.) Instead, you must provide a global definition for it elsewhere, outside the class. This is done by redeclaring the **static** variable using the scope resolution operator to identify the class to which it belongs. This causes storage for the variable to be allocated. (Remember, a class declaration is simply a logical construct that does not have physical reality.)

To understand the usage and effect of a **static** data member, consider this program:

```
#include <iostream>
using namespace std;
class shared {
static int a;
int b;
public:
void set(int i, int j) {a=i; b=j;}
void show();
} ;
int shared::a; // define a
void shared::show()
{
cout << "This is static a: " << a;
cout << "\nThis is non-static b: " << b;
cout << "\n";
}
int main()
{
```

```
shared x, y;
x.set(1, 1); // set a to 1
x.show();
y.set(2, 2); // change a to 2
y.show();
x.show(); /* Here, a has been changed for both x and y because a is shared by both objects. */
return 0;
}
```

This program displays the following output when run.

```
This is static a: 1
This is non-static b: 1
This is static a: 2
This is non-static b: 2
This is static a: 2
This is non-static b: 1
```

Notice that the integer **a** is declared both inside **shared** and outside of it. As mentioned earlier, this is necessary because the declaration of **a** inside **shared** does not allocate storage. *As a convenience, older versions of C++ did not require the second declaration of a **static** member variable. However, this convenience gave rise to serious inconsistencies and it was eliminated several years ago. However, you may still find older C++ code that does not redeclare **static** member variables. In these cases, you will need to add the required definitions.*

A **static** member variable exists *before* any object of its class is created. For example, in the following short program, **a** is both **public** and **static**. Thus it may be directly accessed in **main( )**. Further, since **a** exists before an object of **shared** is created, **a** can be given a value at any time. As this program illustrates, the value of **a** is unchanged by the creation of object **x**. For this reason, both output statements display the same value: 99.

```
#include <iostream>
using namespace std;
class shared {
public:
```

```
static int a;
} ;
int shared::a; // define a
int main()
{
// initialize a before creating any objects
shared::a = 99;
cout << "This is initial value of a: " << shared::a;
cout << "\n";
shared x;
cout << "This is x.a: " << x.a;
return 0;
}
```

Notice how **a** is referred to through the use of the class name and the scope resolution operator. In general, to refer to a **static** member independently of an object, you must qualify it by using the name of the class of which it is a member. One use of a **static** member variable is to provide access control to some shared resource used by all objects of a class. For example, you might create several objects, each of which needs to write to a specific disk file. Clearly, however, only one object can be allowed to write to the file at a time. In this case, you will want to declare a **static** variable that indicates when the file is in use and when it is free. Each object then interrogates this variable before writing to the file.

The following program shows how you might use a **static** variable of this type to control access to a scarce resource:

```
#include <iostream>
using namespace std;
class cl {
static int resource;
public:
int get_resource();
void free_resource() {resource = 0;}
```

```
};
int cl::resource; // define resource
int cl::get_resource()
{
if(resource) return 0; // resource already in use
else {
resource = 1;
return 1; // resource allocated to this object
}
}
int main()
{
cl ob1, ob2;
if(ob1.get_resource()) cout << "ob1 has resource\n";
if(!ob2.get_resource()) cout << "ob2 denied resource\n";
ob1.free_resource(); // let someone else use it
if(ob2.get_resource())
cout << "ob2 can now use resource\n";
return 0;
}
```

Another interesting use of a **static** member variable is to keep track of the number of objects of a particular class type that are in existence.

For example,

```
#include <iostream>
using namespace std;
class Counter {
public:
static int count;
Counter() { count++; }
~Counter() { count--; }
};
```

```
int Counter::count;
void f();
int main(void)
{
Counter o1;
cout << "Objects in existence: ";
cout << Counter::count << "\n";
Counter o2;
cout << "Objects in existence: ";
cout << Counter::count << "\n";
f();
cout << "Objects in existence: ";
cout << Counter::count << "\n";
return 0;
}
void f()
{
Counter temp;
cout << "Objects in existence: ";
cout << Counter::count << "\n";
// temp is destroyed when f() returns
}
```

This program produces the following output.

```
Objects in existence: 1
Objects in existence: 2
Objects in existence: 3
Objects in existence: 2
```

As you can see, the **static** member variable **count** is incremented whenever an object is created and decremented when an object is destroyed. This way, it keeps track of how many objects of type **Counter** are currently in existence. By using **static** member variables, you should be able

to virtually eliminate any need for global variables. The trouble with global variables relative to OOP is that they almost always violate the principle of encapsulation.

## 2.11 Functions

**Static Member Functions**

Member functions may also be declared as **static**. There are several restrictions placed on **static** member functions. They may only directly refer to other **static** members of the class. (Of course, global functions and data may be accessed by **static** member functions.) A **static** member function does not have a **this** pointer. (See Chapter 13 for information on **this**.) There cannot be a **static** and a non-**static** version of the same function. A **static** member function may not be virtual. Finally, they cannot be declared as **const** or **volatile**. Following is a slightly reworked version of the shared-resource program from the previous section. Notice that **get_resource( )** is now declared as **static**. As the program illustrates, **get_resource( )** may be called either by itself, independently of any object, by using the class name and the scope resolution operator, or in connection with an object.

```
#include <iostream>
using namespace std;
class cl {
static int resource;
public:
static int get_resource();
void free_resource() { resource = 0; }
};
int cl::resource; // define resource
int cl::get_resource()
{
if(resource) return 0; // resource already in use
else {
resource = 1;
return 1; // resource allocated to this object
```

```
}
}
int main()
{
cl ob1, ob2;
/* get_resource() is static so may be called independent
of any object. */
if(cl::get_resource()) cout << "ob1 has resource\n";
if(!cl::get_resource()) cout << "ob2 denied resource\n";
ob1.free_resource();
if(ob2.get_resource()) // can still call using object syntax
cout << "ob2 can now use resource\n";
return 0;
}
```

Actually, **static** member functions have limited applications, but one good use for them is to "preinitialize" private **static** data before any object is actually created.

For example, this is a perfectly valid C++ program:

```
#include <iostream>
using namespace std;
class static_type {
static int i;
public:
static void init(int x) {i = x;}
void show() {cout << i;}
};
int static_type::i; // define i
int main()
{
// init static data before object creation
static_type::init(100);
```

```
static_type x;
x.show(); // displays 100
return 0;
}
```

# UNIT 3

## Classes & Objects –II

**3.1 Friend functions**

**3.2 Passing objects as arguments**

**3.3 Returning objects**

**3.4 Arrays of objects**

**3.5 Dynamic objects**

**3.6 Pointers to objects**

**3.7 Copy constructors**

**3.8 Generic functions and classes**

**3.9 Applications**

**3.10 Operator overloading using friend functions such as +, - ,**

**Pre-increment, post- increment, [ ] etc., overloading <<, >>.**

## 3.1 Friend functions

**Friend Functions**

It is possible to grant a nonmember function access to the private members of a class by using a **friend**. A **friend** function has access to all **private** and **protected** members of the class for which it is a **friend**. To declare a **friend** function, include its prototype within the class, preceding it with the keyword **friend**. Consider this program:

```
#include <iostream>
using namespace std;
class myclass {
int a, b;
public:
friend int sum(myclass x);
void set_ab(int i, int j);
};
void myclass::set_ab(int i, int j)
{
a = i;
b = j;
}
// Note: sum() is not a member function of any class.
int sum(myclass x)
{
/* Because sum() is a friend of myclass, it can
directly access a and b. */
return x.a + x.b;
}
int main()
{
myclass n;
n.set_ab(3, 4);
cout << sum(n);
```

```
return 0;
}
```

In this example, the **sum( )** function is not a member of **myclass**. However, it still has full access to its private members. Also, notice that **sum( )** is called without the use of the dot operator. Because it is not a member function, it does not need to be (indeed, it may not be) qualified with an object's name. Although there is nothing gained by making **sum( )** a **friend** rather than a member function of **myclass**, there are some circumstances in which **friend** functions are quite valuable.

First, friends can be useful when you are overloading certain types of operators (see Chapter 14). Second, **friend** functions make the creation of some types of I/O functions easier (see Chapter 18). The third reason that **friend** functions may be desirable is that in some cases, two or more classes may contain members that are interrelated relative to other parts of your program. Let's examine this third usage now. To begin, imagine two different classes, each of which displays a pop-up message on the screen when error conditions occur. Other parts of your program may wish to know if a pop-up message is currently being displayed before writing to the screen so that no message is accidentally overwritten. Although you can create member functions in each class that return a value indicating whether a message is active, this means additional overhead when the condition is checked (that is, two function calls, not just one). If the condition needs to be checked frequently, this additional overhead may not be acceptable. However, using a function that is a **friend** of each class, it is possible to check the status of each object by calling only this one function. Thus, in situations like this, a **friend** function allows you to generate more efficient code.

The following program illustrates this concept:

```
#include <iostream>
using namespace std;
const int IDLE = 0;
const int INUSE = 1;
class C2; // forward declaration
class C1 {
```

```
int status; // IDLE if off, INUSE if on screen
// ...
public:
void set_status(int state);
friend int idle(C1 a, C2 b);
};
class C2 {
int status; // IDLE if off, INUSE if on screen
// ...
public:
void set_status(int state);
friend int idle(C1 a, C2 b);
};
void C1::set_status(int state)
{
status = state;
}
void C2::set_status(int state)
{
status = state;
}
int idle(C1 a, C2 b)
{
if(a.status || b.status) return 0;
else return 1;
}
int main()
{
C1 x; C2 y;
x.set_status(IDLE);
```

```
y.set_status(IDLE);
if(idle(x, y)) cout << "Screen can be used.\n";
else cout << "In use.\n";
x.set_status(INUSE);
if(idle(x, y)) cout << "Screen can be used.\n";
else cout << "In use.\n";
return 0;
}
```

Notice that this program uses a *forward declaration* (also called a *forward reference*) for the class **C2**. This is necessary because the declaration of **idle( )** inside **C1** refers to **C2** before it is declared. To create a forward declaration to a class, simply use the form shown in this program. A **friend** of one class may be a member of another.

For example, here is the preceding program rewritten so that **idle( )** is a member of **C1**:

```
#include <iostream>
using namespace std;
const int IDLE = 0;
const int INUSE = 1;
class C2; // forward declaration
class C1 {
int status; // IDLE if off, INUSE if on screen
// ...
public:
void set_status(int state);
int idle(C2 b); // now a member of C1
};
class C2 {
int status; // IDLE if off, INUSE if on screen
// ...
public:
void set_status(int state);
```

```
friend int C1::idle(C2 b);
};
void C1::set_status(int state)
{
status = state;
}
void C2::set_status(int state)
{
status = state;
}
// idle() is member of C1, but friend of C2
int C1::idle(C2 b)
{
if(status || b.status) return 0;
else return 1;
}
int main()
{
C1 x; C2 y;
x.set_status(IDLE);
y.set_status(IDLE);
if(x.idle(y)) cout << "Screen can be used.\n";
else cout << "In use.\n";
x.set_status(INUSE);
if(x.idle(y)) cout << "Screen can be used.\n";
else cout << "In use.\n";
return 0;
}
```

Because **idle( )** is a member of **C1**, it can access the **status** variable of objects of type **C1** directly. Thus, only objects of type **C2** need be passed to **idle( )**. There are two important

restrictions that apply to **friend** functions. First, a derived class does not inherit **friend** functions. Second, **friend** functions may not have a storage-class specifier. That is, they may not be declared as **static** or **extern.**

## 3.2 Passing objects as arguments

**Passing Objects to Functions**

Objects may be passed to functions in just the same way that any other type of variable can. Objects are passed to functions through the use of the standard call-byvalue mechanism. Although the passing of objects is straightforward, some rather unexpected events occur that relate to constructors and destructors.

To understand why, consider this short program.

```
// Passing an object to a function.
#include <iostream>
using namespace std;
class myclass {
int i; public:
myclass(int n);
~myclass();
void set_i(int n) { i=n; }
int get_i() { return i; }
};
myclass::myclass(int n)
{
i = n;
cout << "Constructing " << i << "\n";
}
myclass::~myclass()
{
cout << "Destroying " << i << "\n";
}
```

```
void f(myclass ob);
int main()
{
myclass o(1);
f(o);
cout << "This is i in main: ";
cout << o.get_i() << "\n";
return 0;
}
void f(myclass ob)
{
ob.set_i(2)
cout << "This is local i: " << ob.get_i();
cout << "\n";
}
```

This program produces this output:

```
Constructing 1
This is local i: 2
Destroying 2
This is i in main: 1
Destroying 1
```

As the output shows, there is one call to the constructor, which occurs when **o** is created in **main( )**, but there are *two* calls to the destructor. Let's see why this is the case. When an object is passed to a function, a copy of that object is made (and this copy becomes the parameter in the function). This means that a new object comes into existence. When the function terminates, the copy of the argument (i.e., the parameter) is destroyed. This raises two fundamental questions: First, is the object's constructor called when the copy is made? Second, is the object's destructor called when the copy is destroyed? The answers may, at first, surprise you. When a copy of an argument is made during a function call, the normal constructor is *not* called. Instead, the object's *copy constructor* is called. A copy constructor defines how a copy of an object is made. As explained in Chapter 14, you can explicitly define a copy constructor

for a class that you create . However, if a class does not explicitly define a copy constructor, as is the case here, then C++ provides one by default.

The default copy constructor creates a bitwise (that is, identical) copy of the object. The reason a bitwise copy is made is easy to understand if you think about it. Since a normal constructor is used to initialize some aspect of an object, it must not be called to make a copy of an already existing object. Such a call would alter the contents of the object. When passing an object to a function, you want to use the current state of the object, not its initial state. However, when the function terminates and the copy of the object used as an argument is destroyed, the destructor *is* called. This is necessary because the object has gone out of scope. This is why the preceding program had two calls to the destructor. The first was when the parameter to **f( )** went out-of-scope. The second is when **o** inside **main( )** was destroyed when the program ended.

To summarize: When a copy of an object is created to be used as an argument to a function, the normal constructor is not called. Instead, the default copy constructor makes a bit-by-bit identical copy. However, when the copy is destroyed (usually by going out of scope when the function returns), the destructor is called. Because the default copy constructor creates an exact duplicate of the original, it can, at times, be a source of trouble. Even though objects are passed to functions by means of the normal call-by-value parameter passing mechanism which, in theory, protects and insulates the calling argument, it is still possible for a side effect to occur that may affect, or even damage, the object used as an argument. For example, if an object used as an argument allocates memory and frees that memory when it is destroyed, then its local copy inside the function will free the same memory when its destructor is called. This will leave the original object damaged and effectively useless. To prevent this type of problem you will need to define the copy operation by creating a copy constructor for the class, as explained

## 3.3 Returning objects

**Returning Objects**

A function may return an object to the caller. For example, this is a valid C++ program:

// Returning objects from a function.

```
#include <iostream>
using namespace std;
class myclass {
int i;
public:
void set_i(int n) { i=n; }
int get_i() { return i; }
};
myclass f(); // return object of type myclass
int main()
{
myclass o;
o = f();
cout << o.get_i() << "\n";
return 0;
}
myclass f()
{
myclass x;
x.set_i(1);
return x;
}
```

When an object is returned by a function, a temporary object is automatically created that holds the return value. It is this object that is actually returned by the function. After the value has been returned, this object is destroyed. The destruction of this temporary object may cause unexpected side effects in some situations. For example, if the object returned by the function has a destructor that frees dynamically allocated memory, that memory will be freed even though the object that is receiving the return value is still using it. There are ways to overcome this problem that involve overloading the assignment operator (see Chapter 15) and defining a copy constructor.

## 3.4 Arrays of objects

**Arrays of Objects**

In C++, it is possible to have arrays of objects. The syntax for declaring and using an object array is exactly the same as it is for any other type of array. For example, this program uses a three-element array of objects:

```
#include <iostream>
using namespace std;
class cl {
int i;
public:
void set_i(int j) { i=j; }
int get_i() { return i; }
};
int main()
{
cl ob[3];
int i;
for(i=0; i<3; i++) ob[i].set_i(i+1);
for(i=0; i<3; i++)
cout << ob[i].get_i() << "\n";
return 0;
}
```

This program displays the numbers **1**, **2**, and **3** on the screen.

If a class defines a parameterized constructor, you may initialize each object in an array by specifying an initialization list, just like you do for other types of arrays. However, the exact form of the initialization list will be decided by the number of parameters required by the object's constructors. For objects whose constructors have only one parameter, you can simply specify a list of initial values, using the normal array-initialization syntax. As each element in the array is created, a value from the list is passed to the constructor's parameter.

For example, here is a slightly different version of the preceding program that uses an initialization:

```
#include <iostream>
using namespace std;
class cl {
int i;
public:
cl(int j) { i=j; } // constructor
int get_i() { return i; }
};
int main()
{
cl ob[3] = {1, 2, 3}; // initializers
int i;
for(i=0; i<3; i++)
cout << ob[i].get_i() << "\n";
return 0;
}
```

As before, this program displays the numbers **1**, **2**, and **3** on the screen.

Actually, the initialization syntax shown in the preceding program is shorthand for this longer form: cl ob[3] = { cl(1), cl(2), cl(3) };

Here, the constructor for **cl** is invoked explicitly. Of course, the short form used in the program is more common. The short form works because of the automatic conversion that applies to constructors taking only one argument. Thus, the short form can only be used to initialize object arrays whose constructors only require one argument. If an object's constructor requires two or more arguments, you will have to use the longer initialization form.

For example,

```
#include <iostream>
using namespace std;
class cl {
int h;
```

```
int i;
public:
cl(int j, int k) { h=j; i=k; } // constructor with 2 parameters
int get_i() {return i;}
int get_h() {return h;}
};
int main()
{
cl ob[3] = {
cl(1, 2), // initialize
cl(3, 4),
cl(5, 6)
};
int i;
for(i=0; i<3; i++) {
cout << ob[i].get_h();
cout << ", ";
cout << ob[i].get_i() << "\n";
}
return 0;
}
```

Here, **cl**'s constructor has two parameters and, therefore, requires two arguments. This means that the shorthand initialization format cannot be used and the long form, shown in the example, must be employed.

## 3.5 Dynamic objects

A special case situation occurs if you intend to create both initialized and uninitialized arrays of objects. Consider the following **class.**

```
class cl {
int i;
public:
```

```
cl(int j) { i=j; }
int get_i() { return i; }
};
```

Here, the constructor defined by **cl** requires one parameter. This implies that any array declared of this type must be initialized. That is, it precludes this array declaration: cl a[9]; // error, constructor requires initializers The reason that this statement isn't valid (as **cl** is currently defined) is that it implies that **cl** has a parameterless constructor because no initializers are specified. However, as it stands, **cl** does not have a parameterless constructor. Because there is no valid constructor that corresponds to this declaration, the compiler will report an error.

To solve this problem, you need to overload the constructor, adding one that takes no parameters, as shown next. In this way, arrays that are initialized and those that are not are both allowed.

```
class cl {
int i;
public:
cl() { i=0; } // called for non-initialized arrays
cl(int j) { i=j; } // called for initialized arrays
int get_i() { return i; }
};
```

Given this **class**, both of the following statements are permissible:

```
cl a1[3] = {3, 5, 6}; // initialized
cl a2[34]; // uninitialized
```

## 3.6 Pointers to objects

**Pointers to Objects**

Just as you can have pointers to other types of variables, you can have pointers to objects. When accessing members of a class given a pointer to an object, use the arrow (–>) operator instead of the dot operator.

The next program illustrates how to access an object given a pointer to it:

```
#include <iostream>
using namespace std;
class cl {
int i;
public:
cl(int j) { i=j; }
int get_i() { return i; }
};
int main()
{
cl ob(88), *p;
p = &ob; // get address of ob
cout << p->get_i(); // use -> to call get_i()
return 0;
}
```

As you know, when a pointer is incremented, it points to the next element of its type. For example, an integer pointer will point to the next integer. In general, all pointer arithmetic is relative to the base type of the pointer. (That is, it is relative to the type of data that the pointer is declared as pointing to.) The same is true of pointers to objects.

For example, this program uses a pointer to access all three elements of array **ob** after being assigned **ob**'s starting address:

```
#include <iostream>
using namespace std;
class cl {
int i;
public:
cl() { i=0; }
cl(int j) { i=j; }
int get_i() { return i; }
};
```

```
int main()
{
cl ob[3] = {1, 2, 3}
cl *p;
int i;
p = ob; // get start of array
for(i=0; i<3; i++) {
cout << p->get_i() << "\n";
p++; // point to next object
}
return 0;
}
```

You can assign the address of a public member of an object to a pointer and then access that member by using the pointer. For example, this is a valid C++ program that displays the number **1** on the screen:

```
#include <iostream>
using namespace std;
class cl {
public:
int i;
cl(int j) { i=j; }
};
int main()
{
cl ob(1);
int *p;
p = &ob.i; // get address of ob.i
cout << *p; // access ob.i via p
return 0;
}
```

Because **p** is pointing to an integer, it is declared as an integer pointer. It is irrelevant that **i** is a member of **ob** in this situation.

## 3.7 Copy constructors

**Copy Constructors**

One of the more important forms of an overloaded constructor is the *copy constructor*. Defining a copy constructor can help you prevent problems that might occur when one object is used to initialize another. Let's begin by restating the problem that the copy constructor is designed to solve. By default, when one object is used to initialize another, C++ performs a bitwise copy. That is, an identical copy of the initializing object is created in the target object. Although this is perfectly adequate for many cases—and generally exactly what you want to happen—there are situations in which a bitwise copy should not be used. One of the most common is when an object allocates memory when it is created.

For example, assume a class called *MyClass* that allocates memory for each object when it is created, and an object *A* of that class. This means that *A* has already allocated its memory. Further, assume that *A* is used to initialize *B*,
as shown here: MyClass B = A;
If a bitwise copy is performed, then *B* will be an exact copy of *A*. This means that *B* will be using the same piece of allocated memory that *A* is using, instead of allocating its own. Clearly, this is not the desired outcome. For example, if *MyClass* includes a destructor that frees the memory, then the same piece of memory will be freed twice when *A* and *B* are destroyed! The same type of problem can occur in two additional ways: first, when a copy of an object is made when it is passed as an argument to a function; second, when a temporary object is created as a return value from a function. Remember, temporary objects are automatically created to hold the return value of a function and they may also be created in certain other circumstances. To solve the type of problem just described, C++ allows you to create a copy constructor, which the compiler uses when one object initializes another. Thus, your copy constructor bypasses the default bitwise copy.

The most common general form

of a copy constructor is

classname (const *classname &o*) {

// body of constructor

}

Here, *o* is a reference to the object on the right side of the initialization. It is permissible for a copy constructor to have additional parameters as long as they have default arguments defined for them. However, in all cases the first parameter must be a reference to the object doing the initializing. It is important to understand that C++ defines two distinct types of situations in which the value of one object is given to another. The first is assignment. The second is initialization, which can occur any of three ways:

■ When one object explicitly initializes another, such as in a declaration

■ When a copy of an object is made to be passed to a function

■ When a temporary object is generated (most commonly, as a return value)

The copy constructor applies only to initializations. For example, assuming a class called **myclass**, and that **y** is an object of type **myclass**, each of the following statements involves initialization.

myclass x = y; // y explicitly initializing x

func(y); // y passed as a parameter

y = func(); // y receiving a temporary, return object

Following is an example where an explicit copy constructor is needed. This program creates a very limited "safe" integer array type that prevents array boundaries from being overrun. (Chapter 15 shows a better way to create a safe array that uses overloaded operators.) Storage for each array is allocated by the use of **new**, and a pointer to the memory is maintained within each array object.

```
/* This program creates a "safe" array class. Since space for the array is allocated using new, a copy constructor is provided to allocate memory when one array object is used to initialize another. */
#include <iostream>
#include <new>
#include <cstdlib>
using namespace std;
```

```
class array {
int *p; int
size; public:
array(int sz) {
try {
p = new int[sz];
} catch (bad_alloc xa) {
cout << "Allocation Failure\n";
exit(EXIT_FAILURE);
}
size = sz;
}
~array() { delete [] p; }
// copy constructor
array(const array &a);
void put(int i, int j) {
if(i>=0 && i<size) p[i] = j;
}
int get(int i) {
return p[i];
}
};
// Copy Constructor
array::array(const array &a) {
int i;
try {
p = new int[a.size];
} catch (bad_alloc xa) {
cout << "Allocation Failure\n";
exit(EXIT_FAILURE);
```

```
}
for(i=0; i<a.size; i++) p[i] = a.p[i];
}
int main()
{
array num(10);
int i;
for(i=0; i<10; i++) num.put(i, i);
for(i=9; i>=0; i--) cout << num.get(i);
cout << "\n";
// create another array and initialize with num
array x(num); // invokes copy constructor
for(i=0; i<10; i++) cout << x.get(i);
return 0;
}
```

Let's look closely at what happens when **num** is used to initialize **x** in the statement array x(num); // invokes copy constructor The copy constructor is called, memory for the new array is allocated and stored in **x.p**, and the contents of **num** are copied to **x**'s array. In this way, **x** and **num** have arrays that contain the same values, but each array is separate and distinct. (That is, **num.p** and **x.p** do not point to the same piece of memory.) If the copy constructor had not been created, the default bitwise initialization would have resulted in **x** and **num** sharing the same memory for their arrays. (That is, **num.p** and **x.p** would have indeed pointed to the same location.) Remember that the copy constructor is called only for initializations.

For example, this sequence does not call the copy constructor defined in the preceding program:

```
array a(10);
// ...
array b(10);
b = a; // does not call copy constructor
```

In this case, **b** = **a** performs the assignment operation. If = is not overloaded (as it is not here), a bitwise copy will be made. Therefore, in some cases, you may need to overload the = operator as well as create a copy constructor to avoid certain types of problems.

## 3.8 Generic functions and classes

**Generic Functions**

Ageneric function defines a general set of operations that will be applied to various types of data. The type of data that the function will operate upon is passed to it as a parameter. Through a generic function, a single general procedure can be applied to a wide range of data. As you probably know, many algorithms are logically the same no matter what type of data is being operated upon. For example, the Quicksort sorting algorithm is the same whether it is applied to an array of integers or an array of floats. It is just that the type of the data being sorted is different. By creating a generic function, you can define the nature of the algorithm, independent of any data. Once you have done this, the compiler will automatically generate the correct code for the type of data that is actually used when you execute the function. In essence, when you create a generic function you are creating a function that can automatically overload itself.

A generic function is created using the keyword **template**. The normal meaning of the word "template" accurately reflects its use in C++. It is used to create a template (or framework) that describes what a function will do, leaving it to the compiler to fill in the details as needed.

The general form of a template function definition is shown here: template <class *Ttype*> *ret-type func-name*(*parameter list*)

{

// *body of function*

}

Here, *Ttype* is a placeholder name for a data type used by the function. This name may be used within the function definition. However, it is only a placeholder that the compiler will automatically replace with an actual data type when it creates a specific version of the function.

Although the use of the keyword **class** to specify a generic type in a **template** declaration is traditional, you may also use the keyword **typename**.

The following example creates a generic function that swaps the values of the two variables with which it is called. Because the general process of exchanging two values is independent of the type of the variables, it is a good candidate for being made into a generic function.

```
// Function template example.
#include <iostream>
using namespace std;
// This is a function template.
template <class X> void swapargs(X &a, X &b)
{
X temp;
temp = a;
a = b;
b = temp;
}
int main()
{
int i=10, j=20;
double x=10.1, y=23.3;
char a='x', b='z';
cout << "Original i, j: " << i << ' ' << j << '\n';
cout << "Original x, y: " << x << ' ' << y << '\n';
cout << "Original a, b: " << a << ' ' << b << '\n';
swapargs(i, j); // swap integers
swapargs(x, y); // swap floats
swapargs(a, b); // swap chars
cout << "Swapped i, j: " << i << ' ' << j << '\n';
cout << "Swapped x, y: " << x << ' ' << y << '\n';
cout << "Swapped a, b: " << a << ' ' << b << '\n';
```

```
return 0;
}
```

Let's look closely at this program. The line:

template <class X> void swapargs(X &a, X &b)

tells the compiler two things: that a template is being created and that a generic definition is beginning. Here, **X** is a generic type that is used as a placeholder. After the **template** portion, the function **swapargs( )** is declared, using **X** as the data type of the values that will be swapped. In **main( )**, the **swapargs( )** function is called using three different types of data: **int**s, **double**s, and **char**s. Because **swapargs( )** is a generic function, the compiler automatically creates three versions of **swapargs( )**: one that will exchange integer values, one that will exchange floating-point values, and one that will swap characters.

Here are some important terms related to templates. First, a generic function (that is, a function definition preceded by a **template** statement) is also called a *template function*. Both terms will be used interchangeably in this book. When the compiler creates a specific version of this function, it is said to have created a *specialization.* This is also called a *generated function*. The act of generating a function is referred to as *instantiating* it. Put differently, a generated function is a specific instance of a template function. Since C++ does not recognize end-of-line as a statement terminator, the **template** clause of a generic function definition does not have to be on the same line as the function's name.

The following example shows another common way to format the **swapargs( )** function.

```
template <class X>
void swapargs(X &a, X &b)
{
X temp;
temp = a;
a = b;
b = temp;
}
```

If you use this form, it is important to understand that no other statements can occur between the **template** statement and the start of the generic function definition.

For example, the fragment shown next will not compile.

```
// This will not compile.
template <class X>
int i; // this is an error
void swapargs(X &a, X &b)
{
X temp;
temp = a;
a = b;
b = temp;
}
```

As the comments imply, the **template** specification must directly precede the function definition.

**A Function with Two Generic Types**

You can define more than one generic data type in the **template** statement by using a comma-separated list. For example, this program creates a template function that has two generic types.

```
#include <iostream>
using namespace std;
template <class type1, class type2>
void myfunc(type1 x, type2 y)
{
cout << x << ' ' << y << '\n';
}
int main()
{
myfunc(10, "I like C++");
myfunc(98.6, 19L);
```

```
return 0;
}
```

In this example, the placeholder types **type1** and **type2** are replaced by the compiler with the data types **int** and **char ***, and **double** and **long**, respectively, when the compiler generates the specific instances of **myfunc( )** within **main( )**.

## 3.9 Applications

**Applying Generic Functions**

Generic functions are one of C++'s most useful features. They can be applied to all types of situations. As mentioned earlier, whenever you have a function that defines a generalizable algorithm, you can make it into a template function. Once you have done so, you may use it with any type of data without having to recode it. Before moving on to generic classes, two examples of applying generic functions will be given. They illustrate how easy it is to take advantage of this powerful C++ feature.

**A Generic Sort**

Sorting is exactly the type of operation for which generic functions were designed. Within wide latitude, a sorting algorithm is the same no matter what type of data is being sorted. The following program illustrates this by creating a generic bubble sort. While the bubble sort is a rather poor sorting algorithm, its operation is clear and uncluttered and it makes an easy-to-understand example.

The **bubble( )** function will sort any type of array. It is called with a pointer to the first element in the array and the number of elements in the array.

```
// A Generic bubble sort.
#include <iostream>
using namespace std;
template <class X> void bubble(
X *items, // pointer to array to be sorted
int count) // number of items in array
{
```

```
register int a, b;
X t;
for(a=1; a<count; a++)
for(b=count-1; b>=a; b--)
if(items[b-1] > items[b]) {
// exchange elements
t = items[b-1];
items[b-1] = items[b];
items[b] = t;
}
}
int main()
{
int iarray[7] = {7, 5, 4, 3, 9, 8, 6};
double darray[5] = {4.3, 2.5, -0.9, 100.2, 3.0};
int i;
cout << "Here is unsorted integer array: ";
for(i=0; i<7; i++)
cout << iarray[i] << ' ';
cout << endl;
cout << "Here is unsorted double array: ";
for(i=0; i<5; i++)
cout << darray[i] << ' ';
cout << endl;
bubble(iarray, 7);
bubble(darray, 5);
cout << "Here is sorted integer array: ";
for(i=0; i<7; i++)
cout << iarray[i] << ' ';
cout << endl;
cout << "Here is sorted double array: ";
```

```
for(i=0; i<5; i++)
cout << darray[i] << ' ';
cout << endl;
return 0;
}
```

The output produced by the program is shown here.

```
Here is unsorted integer array: 7 5 4 3 9 8 6
Here is unsorted double array: 4.3 2.5 -0.9 100.2 3
Here is sorted integer array: 3 4 5 6 7 8 9
Here is sorted double array: -0.9 2.5 3 4.3 100.2
```

As you can see, the preceding program creates two arrays: one integer and one **double**. It then sorts each. Because **bubble( )** is a template function, it is automatically overloaded to accommodate the two different types of data. You might want to try using **bubble( )** to sort other types of data, including classes that you create. In each case, the compiler will create the right version of the function for you.

**Compacting an Array**

Another function that benefits from being made into a template is called **compact( )**. This function compacts the elements in an array. It is not uncommon to want to remove elements from the middle of an array and then move the remaining elements down so that all unused elements are at the end. This sort of operation is the same for all types of arrays because it is independent of the type data actually being operated upon. The generic **compact( )** function shown in the following program is called with a pointer to the first element in the array, the number of elements in the array, and the starting and ending indexes of the elements to be removed. The function then removes those elements and compacts the array. For the purposes of illustration, it also zeroes the unused elements at the end of the array that have been freed by the compaction.

```
// A Generic array compaction function.
#include <iostream>
using namespace std;
```

```
template <class X> void compact(
X *items, // pointer to array to be compacted
int count, // number of items in array
int start, // starting index of compacted region
int end) // ending index of compacted region
{
register int i;
for(i=end+1; i<count; i++, start++)
items[start] = items[i];
/* For the sake of illustration, the remainder of
the array will be zeroed. */
for( ; start<count; start++) items[start] = (X) 0;
}
int main()
{
int nums[7] = {0, 1, 2, 3, 4, 5, 6};
char str[18] = "Generic Functions";
int i;
cout << "Here is uncompacted integer array: ";
for(i=0; i<7; i++)
cout << nums[i] << ' ';
cout << endl;
cout << "Here is uncompacted string: ";
for(i=0; i<18; i++)
cout << str[i] << ' ';
cout << endl;
compact(nums, 7, 2, 4);
compact(str, 18, 6, 10);
cout << "Here is compacted integer array: ";
for(i=0; i<7; i++)
cout << nums[i] << ' ';
```

```
cout << endl;
cout << "Here is compacted string: ";
for(i=0; i<18; i++)
cout << str[i] << ' ';
cout << endl;
return 0;
}
```

This program compacts two different types of arrays. One is an integer array, and the other is a string. However, the **compact( )** function will work for any type of array.

The output from this program in shown here.

```
Here is uncompacted integer array: 0 1 2 3 4 5 6
Here is uncompacted string: G e n e r i c F u n c t i o n s
Here is compacted integer array: 0 1 5 6 0 0 0
Here is compacted string: G e n e r i c t i o n s
```

As the preceding examples illustrate, once you begin to think in terms of templates, many uses will naturally suggest themselves. As long as the underlying logic of a function is independent of the data, it can be made into a generic function.

## 3.10 Operator overloading using friend functions such as +, - , Pre-increment, post- increment, [ ] etc., overloading <<, >>.

### Operator Overloading Using a Friend Function

You can overload an operator for a class by using a nonmember function, which is usually a friend of the class. Since a **friend** function is not a member of the class, it does not have a **this** pointer. Therefore, an overloaded friend operator function is passed the operands explicitly. This means that a friend function that overloads a binary operator has two parameters, and a friend function that overloads a unary operator has one parameter. When overloading a binary operator using a friend function, the left operand is passed in the first parameter and the right operand is passed in the second parameter.

In this program, the **operator+( )** function is made into a friend:

```
#include   <iostream>
using  namespace  std;
class loc {
int longitude, latitude;
public:
loc() {} // needed to construct temporaries
loc(int lg, int lt) {
longitude = lg;
latitude = lt;
}
void show() {
cout << longitude << " ";
cout << latitude << "\n";
}
friend loc operator+(loc op1, loc op2); // now a friend
loc operator-(loc op2);
loc operator=(loc op2);
loc operator++();
};
// Now, + is overloaded using friend function.
loc operator+(loc op1, loc op2)
{
loc temp;
temp.longitude = op1.longitude + op2.longitude;
temp.latitude = op1.latitude + op2.latitude;
return temp;
}
// Overload - for loc.
loc loc::operator-(loc op2)
{
```

```
loc temp;
// notice order of operands
temp.longitude = longitude - op2.longitude;
temp.latitude = latitude - op2.latitude;
return temp;
}
// Overload assignment for loc
loc loc::operator=(loc op2)
{
longitude = op2.longitude;
latitude = op2.latitude;
return *this; // i.e., return object that generated call
}
// Overload ++ for loc.
loc loc::operator++()
{
longitude++;
latitude++;
return *this;
}
int main()
{
loc ob1(10, 20), ob2( 5, 30);
ob1 = ob1 + ob2;
ob1.show();
return 0;
}
```

There are some restrictions that apply to friend operator functions. First, you may not overload the =, **( )**, **[ ]**, or –> operators by using a friend function. Second, as explained in the next section, when overloading the increment or decrement operators, you will need to use a reference parameter when using a friend function.

**Using a Friend to Overload ++ or – –**

If you want to use a friend function to overload the increment or decrement operators, you must pass the operand as a reference parameter. This is because friend functions do not have **this** pointers. Assuming that you stay true to the original meaning of the ++ and – – operators, these operations imply the modification of the operand they operate upon. However, if you overload these operators by using a friend, then the operand is passed by value as a parameter. This means that a friend operator function has no way to modify the operand. Since the friend operator function is not passed a **this** pointer to the operand, but rather a copy of the operand, no changes made to that parameter affect the operand that generated the call. However, you can remedy this situation by specifying the parameter to the friend operator function as a reference parameter. This causes any changes made to the parameter inside the function to affect the operand that generated the call.

For example, this program uses friend functions to overload the prefix versions of ++ and – – operators relative to the **loc** class:

```
#include <iostream>
using namespace std;
class loc {
int longitude, latitude;
public:
loc() {}
loc(int lg, int lt) {
longitude = lg;
latitude = lt;
}
void show() {
cout << longitude << " ";
cout << latitude << "\n";
}
loc operator=(loc op2);
friend loc operator++(loc &op);
```

```
friend loc operator--(loc &op);
};
// Overload assignment for loc.
loc loc::operator=(loc op2)
{
longitude = op2.longitude;
latitude = op2.latitude;
return *this; // i.e., return object that generated call
}
// Now a friend; use a reference parameter.
loc operator++(loc &op)
{
op.longitude++;
op.latitude++;
return op;
}
// Make op-- a friend; use reference.
loc operator--(loc &op)
{
op.longitude--;
op.latitude--;
return op;
}
int main()
{
loc ob1(10, 20), ob2;
ob1.show();
++ob1;
ob1.show(); // displays 11 21
ob2 = ++ob1;
ob2.show(); // displays 12 22
```

```
--ob2;
ob2.show(); // displays 11 21
return 0;
}
```

If you want to overload the postfix versions of the increment and decrement operators using a friend, simply specify a second, dummy integer parameter. For example, this shows the prototype for the **friend,** postfix version of the increment operator relative to **loc.**

```
// friend, postfix version of ++
friend loc operator++(loc &op, int x);
```

**Overloading Some Special Operators**

C++ defines array subscripting, function calling, and class member access as operations. The operators that perform these functions are the **[ ], ( ),** and **–>**, respectively. These rather exotic operators may be overloaded in C++, opening up some very interesting uses. One important restriction applies to overloading these three operators: They must be nonstatic member functions. They cannot be **friend**s.

**Overloading [ ]**

In C++, the **[ ]** is considered a binary operator when you are overloading it. Therefore, the general form of a member **operator[ ]( )** function is as shown here:

```
type class-name::operator[](int i)
{
// . . .
}
```

Technically, the parameter does not have to be of type **int**, but an **operator[ ]( )** function is typically used to provide array subscripting, and as such, an integer value is generally used.

Given an object called **O**, the expression

```
O[3]
```

translates into this call to the **operator[ ]( )** function:

```
O.operator[](3)
```

That is, the value of the expression within the subscripting operators is passed to the **operator[ ]( )** function in its explicit parameter. The **this** pointer will point to **O**, the object that generated the call.

In the following program, **atype** declares an array of three integers. Its constructor initializes each member of the array to the specified values. The overloaded **operator[ ]( )** function returns the value of the array as indexed by the value of its parameter.

```
#include <iostream>
using namespace std;
class atype {
int a[3];
public:
atype(int i, int j, int k) {
a[0] = i;
a[1] = j;
a[2] = k;
}
int operator[](int i) { return a[i]; }
};
int main()
{
atype ob(1, 2, 3);
cout << ob[1]; // displays 2
return 0;
}
```

You can design the **operator[ ]( )** function in such a way that the **[ ]** can be used on both the left and right sides of an assignment statement. To do this, simply specify the return value of **operator[ ]( )** as a reference. The following program makes this change and shows its use:

```
#include <iostream>
using namespace std;
class atype {
```

```
int a[3];
public:
atype(int i, int j, int k) {
a[0] = i;
a[1] = j;
a[2] = k;
}
int &operator[](int i) { return a[i]; }
};
int main()
{
atype ob(1, 2, 3);
cout << ob[1]; // displays 2
cout << " ";
ob[1] = 25; // [] on left of =
cout << ob[1]; // now displays 25
return 0;
}
```

Because **operator[ ]( )** now returns a reference to the array element indexed by **i,** it can be used on the left side of an assignment to modify an element of the array. (Of course, it may still be used on the right side as well.) One advantage of being able to overload the **[ ]** operator is that it allows a means of implementing safe array indexing in C++. As you know, in C++, it is possible to overrun (or underrun) an array boundary at run time without generating a run-time error message. However, if you create a class that contains the array, and allow access to that array only through the overloaded **[ ]** subscripting operator, then you can intercept an out-of-range index. For example, this program adds a range check to the preceding program and proves that it works:

```
// A safe array example.
#include <iostream>
#include <cstdlib>
using namespace std;
```

```
class atype {
int a[3];
public:
atype(int i, int j, int k) {
a[0] = i;
a[1] = j;
a[2] = k;
}
int &operator[](int i);
};
// Provide range checking for atype.
int &atype::operator[](int i)
{
if(i<0 || i> 2) {
cout << "Boundary Error\n";
exit(1);
}
return a[i];
}
int main()
{
atype ob(1, 2, 3);
cout << ob[1]; // displays 2
cout << " ";
ob[1] = 25; // [] appears on left
cout << ob[1]; // displays 25
ob[3] = 44; // generates runtime error, 3 out-of-range
return 0;
}
```

In this program, when the statement

```
ob[3] = 44;
```

executes, the boundary error is intercepted by **operator[]( ),** and the program is terminated before any damage can be done. (In actual practice, some sort of error-handling function would be called to deal with the out-of-range condition; the program would not have to terminate.)

**Overloading ( )**

When you overload the **( )** function call operator, you are not, per se, creating a new way to call a function. Rather, you are creating an operator function that can be passed an arbitrary number of parameters.

Let's begin with an example. Given the overloaded
operator function declaration
double operator()(int a, float f, char *s);
and an object **O** of its class, then the statement
O(10, 23.34, "hi");
translates into this call to the **operator( )** function.
O.operator()(10, 23.34, "hi");

In general, when you overload the **( )** operator, you define the parameters that you want to pass to that function. When you use the **( )** operator in your program, the arguments you specify are copied to those parameters. As always, the object that generates the call (**O** in this example) is pointed to by the **this** pointer.

Here is an example of overloading **( )** for the **loc** class. It assigns the value of its two arguments to the longitude and latitude of the object to which it is applied.

```
#include <iostream>
using namespace std;
class loc {
int longitude, latitude;
public:
loc() {}
loc(int lg, int lt) {
longitude = lg;
latitude = lt;
```

```
}
void show() {
cout << longitude << " ";
cout << latitude << "\n";
}
loc operator+(loc op2);
loc operator()(int i, int j);
};
// Overload ( ) for loc.
loc loc::operator()(int i, int j)
{
longitude = i;
latitude = j;
return *this;
}
// Overload + for loc.
loc loc::operator+(loc op2)
{
loc temp;
temp.longitude = op2.longitude + longitude;
temp.latitude = op2.latitude + latitude;
return temp;
}
int main()
{
loc ob1(10, 20), ob2(1, 1);
ob1.show();
ob1(7, 8); // can be executed by itself
ob1.show();
ob1 = ob2 + ob1(10, 10); // can be used in expressions
ob1.show();
```

```
return 0;
}
```

The output produced by the program is shown here.

```
10 20
7 8
11 11
```

Remember, when overloading **( ),** you can use any type of parameters and return any type of value. These types will be dictated by the demands of your programs. You can also specify default arguments.

**Overloading –>**

The **–>** pointer operator, also called the *class member access* operator, is considered a unary operator when overloading.

Its general usage is shown here:

*object->element;*

Here, *object* is the object that activates the call. The **operator–>( )** function must return a pointer to an object of the class that **operator–>( )** operates upon. The *element* must be some member accessible within the object.

The following program illustrates overloading the **–>** by showing the equivalence between **ob.i** and **ob–>i** when **operator–>( )** returns the **this** pointer:

```
#include <iostream>
using namespace std;
class myclass {
public:
int i;
myclass *operator->() {return this;}
};
int main()
{
myclass ob;
ob->i = 10; // same as ob.i
```

```
cout << ob.i << " " << ob->i;
return 0;
}
```

An **operator–>( )** function must be a member of the class upon which it works. **Overloading the Comma Operator** You can overload C++'s comma operator. The comma is a binary operator, and like all overloaded operators, you can make an overloaded comma perform any operation you want. However, if you want the overloaded comma to perform in a fashion similar to its normal operation, then your version must discard the values of all operands except the rightmost. The rightmost value becomes the result of the comma operation. This is the way the comma works by default in C++.

Here is a program that illustrates the effect of overloading the comma operator.

```
#include <iostream>
using namespace std;
class loc {
int longitude, latitude;
public:
loc() {}
loc(int lg, int lt) {
longitude = lg;
latitude = lt;
}
void show() {
cout << longitude << " ";
cout << latitude << "\n";
}
loc operator+(loc op2);
loc operator,(loc op2);
};
// overload comma for loc
loc loc::operator,(loc op2)
```

```
{
loc temp;
temp.longitude = op2.longitude;
temp.latitude = op2.latitude;
cout << op2.longitude << " " << op2.latitude << "\n";
return temp;
}
// Overload + for loc
loc loc::operator+(loc op2)
{
loc temp;
temp.longitude = op2.longitude + longitude;
temp.latitude = op2.latitude + latitude;
return temp;
}
int main()
{
loc ob1(10, 20), ob2( 5, 30), ob3(1, 1);
ob1.show();
ob2.show();
ob3.show();
cout << "\n";
ob1 = (ob1, ob2+ob2, ob3);
ob1.show(); // displays 1 1, the value of ob3
return 0;
}
```

This program displays the following output:

```
10 20
5 30
1 1
10 60
```

1 1

1 1

Notice that although the values of the left-hand operands are discarded, each expression is still evaluated by the compiler so that any desired side effects will be performed. Remember, the left-hand operand is passed via **this**, and its value is discarded by the **operator,( )** function. The value of the right-hand operation is returned by the function. This causes the overloaded comma to behave similarly to its default operation. If you want the overloaded comma to do something else, you will have to change these two features.

**Overloading << and >>**

As you know, the << and the >> operators are overloaded in C++ to perform I/O operations on C++'s built-in types. You can also overload these operators so that they perform I/O operations on types that you create. In the language of C++, the << output operator is referred to as the *insertion operator* because it inserts characters into a stream. Likewise, the >> input operator is called the *extraction operator* because it extracts characters from a stream. The functions that overload the insertion and extraction operators are generally called *inserters* and *extractors*, respectively.

**Creating Your Own Inserters**

It is quite simple to create an inserter for a class that you create. All inserter functions have this general form:

```
ostream &operator<<(ostream &stream, class_type obj)
{
// body of inserter
return stream;
}
```

Notice that the function returns a reference to a stream of type **ostream**. (Remember, **ostream** is a class derived from **ios** that supports output.) Further, the first parameter to the function is a reference to the output stream. The second parameter is the object being inserted. (The second parameter may also be a reference to the object being inserted.) The last thing the inserter must do before exiting is return *stream*. This allows the inserter to be used in a larger I/O expression.

Within an inserter function, you may put any type of procedures or operations that you want. That is, precisely what an inserter does is completely up to you. However, for the inserter to be in keeping with good programming practices, you should limit its operations to outputting information to a stream. For example, having an inserter compute pi to 30 decimal places as a side effect to an insertion operation is probably not a very good idea! To demonstrate a custom inserter, one will be created for objects of type

**phonebook**, shown here.

```
class phonebook {
public:
char name[80];
int areacode;
int prefix;
int num;
phonebook(char *n, int a, int p, int nm)
{
strcpy(name, n);
areacode = a;
prefix = p;
num = nm;
}
};
```

This class holds a person's name and telephone number. Here is one way to create an inserter function for objects of type **phonebook**.

```
// Display name and phone number
ostream &operator<<(ostream &stream, phonebook o)
{
stream << o.name << " ";
stream << "(" << o.areacode << ") ";
stream << o.prefix << "-" << o.num << "\n";
return stream; // must return stream
}
```

Here is a short program that illustrates the **phonebook** inserter function:

```
#include <iostream>
#include <cstring>
using namespace std;
class phonebook {
public:
char name[80];
int areacode;
int prefix;
int num;
phonebook(char *n, int a, int p, int nm)
{
strcpy(name, n);
areacode = a;
prefix = p;
num = nm;
}
};
// Display name and phone number.
ostream &operator<<(ostream &stream, phonebook o)
{
stream << o.name << " ";
stream << "(" << o.areacode << ") ";
stream << o.prefix << "-" << o.num << "\n";
return stream; // must return stream
}
int main()
{
phonebook a("Ted", 111, 555, 1234);
phonebook b("Alice", 312, 555, 5768);
phonebook c("Tom", 212, 555, 9991);
```

```
cout << a << b << c;
return 0;
}
```

The program produces this output:

```
Ted (111) 555-1234
Alice (312) 555-5768
Tom (212) 555-9991
```

In the preceding program, notice that the **phonebook** inserter is not a member of **phonebook**. Although this may seem weird at first, the reason is easy to understand. When an operator function of any type is a member of a class, the left operand (passed implicitly through **this**) is the object that generates the call to the operator function. Further, this object is an *object of the class* for which the operator function is a member. There is no way to change this. If an overloaded operator function is a member of a class, the left operand must be an object of that class. However, when you overload inserters, the left operand is a *stream* and the right operand is an object of the class. Therefore, overloaded inserters cannot be members of the class for which they are overloaded.

The variables **name**, **areacode**, **prefix**, and **num** are public in the preceding program so that they can be accessed by the inserter. The fact that inserters cannot be members of the class for which they are defined seems to be a serious flaw in C++. Since overloaded inserters are not members, how can they access the private elements of a class? In the foregoing program, all members were made public. However, encapsulation is an essential component of object-oriented programming. Requiring that all data that will be output be public conflicts with this principle. Fortunately, there is a solution to this dilemma: Make the inserter a **friend** of the class. This preserves the requirement that the first argument to the overloaded inserter be a stream and still grants the function access to the private members of the class for which it is overloaded.

Here is the same program modified to make the inserter into a **friend** function:

```
#include <iostream>
#include <cstring>
```

```
using namespace std;
class phonebook {
// now private
char name[80];
int areacode;
int prefix;
int num;
public:
phonebook(char *n, int a, int p, int nm)
{
strcpy(name, n);
areacode = a;
prefix = p;
num = nm;
}
friend ostream &operator<<(ostream &stream, phonebook o);
};
// Display name and phone number.
ostream &operator<<(ostream &stream, phonebook o)
{
stream << o.name << " ";
stream << "(" << o.areacode << ") ";
stream << o.prefix << "-" << o.num << "\n";
return stream; // must return stream
}
int main()
{
phonebook a("Ted", 111, 555, 1234);
phonebook b("Alice", 312, 555, 5768);
phonebook c("Tom", 212, 555, 9991);
cout << a << b << c;
```

```
return 0;
}
```

When you define the body of an inserter function, remember to keep it as general as possible. For example, the inserter shown in the preceding example can be used with any stream because the body of the function directs its output to **stream**, which is the stream that invoked the inserter. While it would not be technically wrong to have written

```
stream << o.name << " ";
```

as

```
cout << o.name << " ";
```

this would have the effect of hard-coding **cout** as the output stream. The original version will work with any stream, including those linked to disk files. Although in some situations, especially where special output devices are involved, you may want to hard-code the output stream, in most cases you will not. In general, the more flexible your inserters are, the more valuable they are.

*The inserter for the **phonebook** class works fine unless the value of **num** is something like 0034, in which case the preceding zeroes will not be displayed. To fix this, you can either make **num** into a string or you can set the fill character to zero and use the **width( )** format function to generate the leading zeroes. The solution is left to the reader as an exercise.*Before moving on to extractors, let's look at one more example of an inserter function. An inserter need not be limited to handling only text. An inserter can be used to output data in any form that makes sense. For example, an inserter for some class that is part of a CAD system may output plotter instructions. Another inserter might generate graphics images. An inserter for a Windows-based program could display a dialog box. To sample the flavor of outputting things other than text, examine the following program, which draws boxes on the screen. (Because C++ does not define a graphics library, the program uses characters to draw a box, but feel free to substitute graphics if your system supports them.)

```
#include <iostream>
using namespace std;
class box {
int x, y;
public:
```

```
box(int i, int j) { x=i; y=j; }
friend ostream &operator<<(ostream &stream, box o);
};
// Output a box.
ostream &operator<<(ostream &stream, box o)
{
register int i, j;
for(i=0; i<o.x; i++)
stream << "*";
stream << "\n";
for(j=1; j<o.y-1; j++) {
for(i=0; i<o.x; i++)
if(i==0 || i==o.x-1) stream << "*";
else stream << " ";
stream << "\n";
}
for(i=0; i<o.x; i++)
stream << "*";
stream << "\n";
return stream;
}
int main()
{
box a(14, 6), b(30, 7), c(40, 5);
cout << "Here are some boxes:\n";
cout << a << b << c;
return 0;
}
```

The program displays the following:

```
Here are some boxes:
**************
```

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

# UNIT 4

## Inheritance – I

**4.1 Base Class**

**4.2 Inheritance and protected members**

**4.3 Protected base class inheritance**

**4.4 Inheriting multiple base classes**

## 4.1 Base Class

**Base-Class Access Control**

When a class inherits another, the members of the base class become members of the derived class.

Class inheritance uses this general form:

class *derived-class-name : access base-class-name* {

// *body of class*

};

The access status of the base-class members inside the derived class is determined by *access*. The base-class access specifier must be either **public**, **private**, or **protected**. If no access specifier is present, the access specifier is **private** by default if the derived class is a **class**. If the derived class is a **struct**, then **public** is the default in the absence of an explicit access specifier. Let's examine the ramifications of using **public** or **private** access. (The **protected** specifier is examined in the next section.) When the access specifier for a base class is **public**, all public members of the base become public members of the derived class, and all protected members of the base become protected members of the derived class. In all cases, the base's private elements remain private to the base and are not accessible by members of the derived class.

For example, as illustrated in this program, objects of type **derived** can directly access the public members of **base**:

```
#include <iostream>
using namespace std;
class base {
int i, j;
public:
void set(int a, int b) { i=a; j=b; }
void show() { cout << i << " " << j << "\n"; }
};
class derived : public base {
int k;
public:
```

```
derived(int x) { k=x; }
void showk() { cout << k << "\n"; }
};
int main()
{
derived ob(3);
ob.set(1, 2); // access member of base
ob.show(); // access member of base
ob.showk(); // uses member of derived class
return 0;
}
```

When the base class is inherited by using the **private** access specifier, all public and protected members of the base class become private members of the derived class.

For example, the following program will not even compile because both **set( )** and **show( )** are now private elements of **derived**:

```
// This program won't compile.
#include <iostream>
using namespace std;
class base {
int i, j;
public:
void set(int a, int b) { i=a; j=b; }
void show() { cout << i << " " << j << "\n";}
};
// Public elements of base are private in derived.
class derived : private base {
int k;
public:
derived(int x) { k=x; }
void showk() { cout << k << "\n"; }
};
```

```
int main()
{
derived ob(3);
ob.set(1, 2); // error, can't access set()
ob.show(); // error, can't access show()
return 0;
}
```

*When a base class' access specifier is **private**, public and protected members of the base become private members of the derived class. This means that they are still accessible bymembers of the derived class but cannot be accessed by parts of your program that are not members of either the base or derived class.*

## 4.2 Inheritance and protected members

**Inheritance and protected Members**

The **protected** keyword is included in C++ to provide greater flexibility in the inheritance mechanism. When a member of a class is declared as **protected**, that member is not accessible by other, nonmember elements of the program. With one important exception, access to a protected member is the same as access to a private member—it can be accessed only by other members of its class. The sole exception to this is when a protected member is inherited. In this case, a protected member differs substantially from a private one. As explained in the preceding section, a private member of a base class is not accessible by other parts of your program, including any derived class. However, protected members behave differently. If the base class is inherited as **public**, then the base class' protected members become protected members of the derived class and are, therefore, accessible by the derived class. By using **protected**, you can create class members that are private to their class but that can still be inherited and accessed by a derived class. Here is an example:

```
#include <iostream>
using namespace std;
class base { protected:
int i, j; // private to base, but accessible by derived
public:
```

```
void set(int a, int b) { i=a; j=b; }
void show() { cout << i << " " << j << "\n"; }
};
class derived : public base {
int k;
public:
// derived may access base's i and j
void setk() { k=i*j; }
void showk() { cout << k << "\n"; }
};
int main()
{
derived ob;
ob.set(2, 3); // OK, known to derived
ob.show(); // OK, known to derived
ob.setk();
ob.showk();
return 0;
}
```

In this example, because **base** is inherited by **derived** as **public** and because **i** and **j** are declared as **protected**, **derived**'s function **setk( )** may access them. If **i** and **j** had been declared as **private** by **base**, then **derived** would not have access to them, and the program would not compile.

When a derived class is used as a base class for another derived class, any protected member of the initial base class that is inherited (as public) by the first derived class may also be inherited as protected again by a second derived class. For example, this program is correct, and **derived2** does indeed have access to **i** and **j**.

```
#include <iostream>
using namespace std;
class base
{
```

```
protected:
int i, j;
public:
void set(int a, int b) { i=a; j=b; }
void show() { cout << i << " " << j << "\n"; }
};
// i and j inherited as protected.
class derived1 : public base {
int k;
public:
void setk() { k = i*j; } // legal
void showk() { cout << k << "\n"; }
};
// i and j inherited indirectly through derived1.
class derived2 : public derived1 {
int m;
public:
void setm() { m = i-j; } // legal
void showm() { cout << m << "\n"; }
};
int main()
{
derived1 ob1;
derived2 ob2;
ob1.set(2, 3);
ob1.show();
ob1.setk();
ob1.showk();
ob2.set(3, 4);
ob2.show();
ob2.setk();
```

```
ob2.setm();
ob2.showk();
ob2.showm();
return 0;
```

} If, however, **base** were inherited as **private**, then all members of **base** would become private members of **derived1**, which means that they would not be accessible by **derived2**. (However, **i** and **j** would still be accessible by **derived1**.) This situation is illustrated by the following program, which is in error (and won't compile).

The comments describe each error:

```
// This program won't compile.
#include <iostream>
using namespace std;
class base {
protected:
int i, j;
public:
void set(int a, int b) { i=a; j=b; }
void show() { cout << i << " " << j << "\n"; }
};
// Now, all elements of base are private in derived1.
class derived1 : private base {
int k;
public:
// this is legal because i and j are private to derived1
void setk() { k = i*j; } // OK
void showk() { cout << k << "\n"; }
};
// Access to i, j, set(), and show() not inherited.
class derived2 : public derived1 {
int m;
public:
```

```
// illegal because i and j are private to derived1
void setm() { m = i-j; } // Error
void showm() { cout << m << "\n"; }
};
int main()
{
derived1 ob1;
derived2 ob2;
ob1.set(1, 2); // error, can't use set()ob1.show(); // error, can't use show()
ob2.set(3, 4); // error, can't use set()
ob2.show(); // error, can't use show()
return 0;
}
```

*Even though **base** is inherited as **private** by **derived1**, **derived1** still has access to **base**'s **public** and **protected** elements. However, it cannot pass along this privilege.*

## 4.3 Protected base class inheritance

**Protected Base-Class Inheritance**

It is possible to inherit a base class as **protected.** When this is done, all public and protected members of the base class become protected members of the derived class.

For example,

```
#include <iostream>
using namespace std;
class base {
protected:
int i, j; // private to base, but accessible by derived
public:
void setij(int a, int b) { i=a; j=b; }
void showij() { cout << i << " " << j << "\n"; }
};
// Inherit base as protected.
```

```
class derived : protected base{
int k;
public:
// derived may access base's i and j and setij().
void setk() { setij(10, 12); k = i*j; }
// may access showij() here
void showall() { cout << k << " "; showij(); }
};
int main()
{
derived ob;
// ob.setij(2, 3); // illegal, setij() is
// protected member of derived
ob.setk(); // OK, public member of derived
ob.showall(); // OK, public member of derived
// ob.showij(); // illegal, showij() is protected
// member of derived
return 0;
}
```

As you can see by reading the comments, even though **setij( )** and **showij( )** are public members of **base**, they become protected members of **derived** when it is inherited using the **protected** access specifier. This means that they will not be accessible inside **main( )**.

## 4.4 Inheriting multiple base classes

**Inheriting Multiple Base Classes**

It is possible for a derived class to inherit two or more base classes. For example, in this short example, **derived** inherits both **base1** and **base2**.

```
// An example of multiple base classes.
#include <iostream>
using namespace std;
class base1 {
```

```
protected:
int x;
public:
void showx() { cout << x << "\n"; }
};
class base2 {
protected:
int y;
public:
void showy() {cout << y << "\n";}
};
// Inherit multiple base classes.
class derived: public base1, public base2 {
public:
void set(int i, int j) { x=i; y=j; }
};
int main()
{
derived ob;
ob.set(10, 20); // provided by derived
ob.showx(); // from base1
ob.showy(); // from base2
return 0;
}
```

As the example illustrates, to inherit more than one base class, use a commaseparated list. Further, be sure to use an access-specifier for each base inherited.

# UNIT 5

## Inheritance – II

**5.1 Constructors, Destructors and Inheritance**

**5.2 Passing parameters to base class constructors**

**5.3 Granting access**

**5.4 Virtual base classes**

## 5.1 Constructors, Destructors and Inheritance

There are two major questions that arise relative to constructors and destructors when inheritance is involved. First, when are base-class and derived-class constructors and destructors called? Second, how can parameters be passed to base-class constructors? This section examines these two important topics.

**When Constructors and Destructors Are Executed**

It is possible for a base class, a derived class, or both to contain constructors and/or destructors. It is important to understand the order in which these functions are executed when an object of a derived class comes into existence and when it goes out of existence.

To begin, examine this short program:

```
#include <iostream>
using namespace std;
class base {
public:
base() { cout << "Constructing base\n"; }
~base() { cout << "Destructing base\n"; }
};
class derived: public base {
public:
derived() { cout << "Constructing derived\n"; }
~derived() { cout << "Destructing derived\n"; }
};
int main()
{
derived ob;
// do nothing but construct and destruct ob
return 0;
}
```

As the comment in **main( )** indicates, this program simply constructs and then destroys an object called **ob** that is of class **derived**.

When executed, this program displays Constructing base

Constructing derived
Destructing derived
Destructing base

As you can see, first **base**'s constructor is executed followed by **derived**'s. Next (because **ob** is immediately destroyed in this program), **derived**'s destructor is called, followed by **base**'s. The results of the foregoing experiment can be generalized. When an object of a derived class is created, the base class' constructor will be called first, followed by the derived class' constructor. When a derived object is destroyed, its destructor is called first, followed by the base class' destructor. Put differently, constructors are executed in their order of derivation. Destructors are executed in reverse order of derivation. If you think about it, it makes sense that constructors are executed in order of derivation. Because a base class has no knowledge of any derived class, any initialization it needs to perform is separate from and possibly prerequisite to any initialization performed by the derived class. Therefore, it must be executed first.

Likewise, it is quite sensible that destructors be executed in reverse order of derivation. Because the base class underlies the derived class, the destruction of the base object implies the destruction of the derived object. Therefore, the derived destructor must be called before the object is fully destroyed. In cases of multiple inheritance (that is, where a derived class becomes the base class for another derived class), the general rule applies: Constructors are called in order of derivation, destructors in reverse order.

For example, this program

```
#include <iostream>
using namespace std;
class base {
public:
base() { cout << "Constructing base\n"; }
~base() { cout << "Destructing base\n"; }
};
class derived1 : public base {
public:
derived1() { cout << "Constructing derived1\n"; }
```

```
~derived1() { cout << "Destructing derived1\n"; }
};
class derived2: public derived1 {
public:
derived2() { cout << "Constructing derived2\n"; }
~derived2() { cout << "Destructing derived2\n"; }
};
int main()
{
derived2 ob;
// construct and destruct ob
return 0;
```

displays this output:

```
Constructing base
Constructing derived1
Constructing derived2
Destructing derived2
Destructing derived1
Destructing base
```

The same general rule applies in situations involving multiple base classes.

For example, this program

```
#include   <iostream>
using  namespace  std;
class base1 {
public:
base1() { cout << "Constructing base1\n"; }
~base1() { cout << "Destructing base1\n"; }
};
class base2 {
public:
base2() { cout << "Constructing base2\n"; }
```

```
~base2() { cout << "Destructing base2\n"; }
};
class derived: public base1, public base2 {
public:
derived() { cout << "Constructing derived\n"; }
~derived() { cout << "Destructing derived\n"; }
};
int main()
{
derived ob;
// construct and destruct ob
return 0;
}
```

produces this output:

```
Constructing base1
Constructing base2
Constructing derived
Destructing derived
Destructing base2
Destructing base1
```

As you can see, constructors are called in order of derivation, left to right, as specified in **derived**'s inheritance list. Destructors are called in reverse order, right to left. This means that had **base2** been specified before **base1** in **derived**'s list, as shown here:

```
class derived: public base2, public base1 {
```

then the output of this program would have looked like this:

```
Constructing base2
Constructing base1
Constructing derived
Destructing derived
Destructing base1
Destructing base2
```

## 5.2 Passing parameters to base class constructors

**Passing Parameters to Base-Class Constructors**

So far, none of the preceding examples have included constructors that require arguments. In cases where only the derived class' constructor requires one or more parameters, you simply use the standard parameterized constructor syntax. However, how do you pass arguments to a constructor in a base class? The answer is to use an expanded form of the derived class's constructor declaration that passes along arguments to one or more base-class constructors. The general form of this expanded derived-class constructor declaration is shown here:

*derived-constructor(arg-list) : base1(arg-list),*
*base2(arg-list),*
*// ...*
*baseN(arg-list)*
{
*// body of derived constructor*
}

Here, *base1* through *baseN* are the names of the base classes inherited by the derived class. Notice that a colon separates the derived class' constructor declaration from the base-class specifications, and that the base-class specifications are separated from each other by commas, in the case of multiple base classes. Consider this program:

```
#include <iostream>
using namespace std;
class base {
protected:
int i;
public:
base(int x) { i=x; cout << "Constructing base\n"; }
~base() { cout << "Destructing base\n"; }
};
class derived: public base {
int j;
public:
```

```
// derived uses x; y is passed along to base.
derived(int x, int y): base(y)
{ j=x; cout << "Constructing derived\n"; }
~derived() { cout << "Destructing derived\n"; }
void show() { cout << i << " " << j << "\n"; }
};
int main()
{
derived ob(3, 4);
ob.show(); // displays 4 3
return 0;
}
```

Here, **derived**'s constructor is declared as taking two parameters, **x** and **y**. However, **derived( )** uses only **x; y** is passed along to **base( ).** In general, the derived class' constructor must declare both the parameter(s) that it requires as well as any required by the base class. As the example illustrates, any parameters required by the base class are passed to it in the base class' argument list specified after the colon.

Here is an example that uses multiple base classes:

```
#include <iostream>
using namespace std;
class base1 {
protected:
int i;
public:
base1(int x) { i=x; cout << "Constructing base1\n"; }
~base1() { cout << "Destructing base1\n"; }
};
class base2 {
protected:
int k;
public:
```

```
base2(int x) { k=x; cout << "Constructing base2\n"; }
~base2() { cout << "Destructing base1\n"; }
};
class derived: public base1, public base2 {
int j;
public:
derived(int x, int y, int z): base1(y), base2(z)
{ j=x; cout << "Constructing derived\n"; }
~derived() { cout << "Destructing derived\n"; }
void show() { cout << i << " " << j << " " << k << "\n"; }
};
int main()
{
derived ob(3, 4, 5);
ob.show(); // displays 4 3 5
return 0;
}
```

It is important to understand that arguments to a base-class constructor are passed via arguments to the derived class' constructor. Therefore, even if a derived class' constructor does not use any arguments, it will still need to declare one if the base class requires it. In this situation, the arguments passed to the derived class are simply passed along to the base.

For example, in this program, the derived class' constructor takes no arguments, but **base1( )** and **base2( )** do:

```
#include <iostream>
using namespace std;
class base1 {
protected:
int i;
public:
base1(int x) { i=x; cout << "Constructing base1\n"; }
~base1() { cout << "Destructing base1\n"; }
```

```
};
class base2 {
protected:
int k;
public:
base2(int x) { k=x; cout << "Constructing base2\n"; }
~base2() { cout << "Destructing base2\n"; }
};
class derived: public base1, public base2 {
public:
/* Derived constructor uses no parameter, but still must be declared as taking them to pass them along to base classes. */
derived(int x, int y): base1(x), base2(y)
{ cout << "Constructing derived\n"; }
~derived() { cout << "Destructing derived\n"; }
void show() { cout << i << " " << k << "\n"; }
};
int main()
{
derived ob(3, 4);
ob.show(); // displays 3 4
return 0;
}
```

A derived class' constructor is free to make use of any and all parameters that it is declared as taking, even if one or more are passed along to a base class. Put differently, passing an argument along to a base class does not preclude its use by the derived class as well. For example, this fragment is perfectly valid:

```
class derived: public base {
int j;
public:
// derived uses both x and y and then passes them to base.
```

```
derived(int x, int y): base(x, y)
{ j = x*y; cout << "Constructing derived\n"; }
```

One final point to keep in mind when passing arguments to base-class constructors: The argument can consist of any expression valid at the time. This includes function calls and variables. This is in keeping with the fact that C++ allows dynamic initialization.

## 5.3 Granting access

**Granting Access**

When a base class is inherited as **private**, all public and protected members of that class become private members of the derived class. However, in certain circumstances, you may want to restore one or more inherited members to their original access specification. For example, you might want to grant certain public members of the base class public status in the derived class even though the base class is inherited as **private**. In Standard C++, you have two ways to accomplish this. First, you can use a **using** statement, which is the preferred way. The **using** statement is designed primarily to support namespaces and is discussed in Chapter 23. The second way to restore an inherited member's access specification is to employ an *access declaration* within the derived class. Access declarations are currently supported by Standard C++, but they are deprecated. This means that they should not be used for new code. Since there are still many, many existing programs that use access declarations, they will be examined here.

An access declaration takes this general form:

*base-class::member*;

The access declaration is put under the appropriate access heading in the derived class' declaration. Notice that no type declaration is required (or, indeed, allowed) in an access declaration.

To see how an access declaration works, let's begin with this short fragment:

```
class base {
public:
int j; // public in base
};
// Inherit base as private.
```

```
class derived: private base {
public:
// here is access declaration
base::j; // make j public again
.
.
.
};
```

Because **base** is inherited as **private** by **derived**, the public member **j** is made a private member of **derived**. However, by including base::j;
as the access declaration under **derived**'s **public** heading, **j** is restored to its public status.
You can use an access declaration to restore the access rights of public and protected members. However, you cannot use an access declaration to raise or lower a member's access status. For example, a member declared as private in a base class cannot be made public by a derived class. (If C++ allowed this to occur, it would destroy its encapsulation mechanism!) The following program illustrates the access declaration; notice how it uses access declarations to restore **j**, **seti( )**, and **geti( )** to **public** status.

```
#include <iostream>
using namespace std;
class base {
int i; // private to base
public:
int j, k;
void seti(int x) { i = x; }
int geti() { return i; }
};
// Inherit base as private.
class derived: private base {
public:
/* The next three statements override
base's inheritance as private and restore j,
```

```
seti(), and geti() to public access. */
base::j; // make j public again - but not k
base::seti; // make seti() public
base::geti; // make geti() public
// base::i; // illegal, you cannot elevate access
int a; // public
};
int main()
{
derived ob;
//ob.i = 10; // illegal because i is private in derived
ob.j = 20; // legal because j is made public in derived
//ob.k = 30; // illegal because k is private in derived
ob.a = 40; // legal because a is public in derived
ob.seti(10);
cout << ob.geti() << " " << ob.j << " " << ob.a;
return 0;
}
```

Access declarations are supported in C++ to accommodate those situations in which most of an inherited class is intended to be made private, but a few members are to retain their public or protected status.

*While Standard C++ still supports access declarations, they are deprecated. This means that they are allowed for now, but they might not be supported in the future. Instead, the standard suggests achieving the same effect by applying the* ***using*** *keyword.*

## 5.4 Virtual base classes

**Virtual Base Classes**

An element of ambiguity can be introduced into a C++ program when multiple base classes are inherited.

For example, consider this incorrect program:

```
// This program contains an error and will not compile.
```

```
#include   <iostream>
using  namespace  std;
class base {
public:
int i;
};
// derived1 inherits base.
class derived1 : public base {
public:
int j;
};
// derived2 inherits base.
class derived2 : public base {
public:
int k;
};
/* derived3 inherits both derived1 and derived2.
This means that there are two copies of base
in derived3! */
class derived3 : public derived1, public derived2 {
public:
int sum;
};
int main()
derived3 ob;
ob.i = 10; // this is ambiguous, which i???
ob.j = 20;
ob.k = 30;
// i ambiguous here, too
ob.sum = ob.i + ob.j + ob.k;
// also ambiguous, which i?
```

```
cout << ob.i << " ";
cout << ob.j << " " << ob.k << " ";
cout << ob.sum;
return 0;
}
```

As the comments in the program indicate, both **derived1** and **derived2** inherit **base**. However, **derived3** inherits both **derived1** and **derived2**. This means that there are two copies of **base** present in an object of type **derived3**. Therefore, in an expression like

ob.i = 10;

which **i** is being referred to, the one in **derived1** or the one in **derived2?** Because there are two copies of **base** present in object **ob**, there are two **ob.i**s! As you can see, the statement is inherently ambiguous.

There are two ways to remedy the preceding program. The first is to apply the scope resolution operator to **i** and manually select one **i.** For example, this version of the program does compile and run as expected:

```
// This program uses explicit scope resolution to select i.
#include <iostream>
using namespace std;
class base {
public:
int i;
};
// derived1 inherits base.
class derived1 : public base {
public:
int j;
};
// derived2 inherits base.
class derived2 : public base {
public:
int k;
```

```
};
/* derived3 inherits both derived1 and derived2.This means that there are two copies of base
in derived3! */
class derived3 : public derived1, public derived2 {
public:
int sum;
};
int main()
{
derived3 ob;
ob.derived1::i = 10; // scope resolved, use derived1's i
ob.j = 20;
ob.k = 30;
// scope resolved
ob.sum = ob.derived1::i + ob.j + ob.k;
// also resolved here
cout << ob.derived1::i << " ";
cout << ob.j << " " << ob.k << " ";
cout << ob.sum;
return 0;
}
```

As you can see, because the **::** was applied, the program has manually selected **derived1**'s version of **base**. However, this solution raises a deeper issue: What if only one copy of **base** is actually required? Is there some way to prevent two copies from being included in **derived3**? The answer, as you probably have guessed, is yes. This solution is achieved using **virtual** base classes.

When two or more objects are derived from a common base class, you can prevent multiple copies of the base class from being present in an object derived from those objects by declaring the base class as **virtual** when it is inherited. You accomplish this by preceding the base class' name with the keyword **virtual** when it is inherited. For example, here is another version of the example program in which **derived3** contains only one copy of **base**:

```
// This program uses virtual base classes.
#include <iostream>
using namespace std;
class base {
public:
int i;
};
// derived1 inherits base as virtual.
class derived1 : virtual public base {
public:
int j;
};
// derived2 inherits base as virtual.
class derived2 : virtual public base {
public:
int k;
};
/* derived3 inherits both derived1 and derived2.
This time, there is only one copy of base class. */
class derived3 : public derived1, public derived2 {
public:
int sum;
};
int main()
{
derived3 ob;
ob.i = 10; // now unambiguous
ob.j = 20;
ob.k = 30;
// unambiguous
ob.sum = ob.i + ob.j + ob.k;
```

```
// unambiguous
cout << ob.i << " ";
cout << ob.j << " " << ob.k << " ";
cout << ob.sum;
return 0;
}
```

As you can see, the keyword **virtual** precedes the rest of the inherited **class**' specification. Now that both **derived1** and **derived2** have inherited **base** as **virtual**, any multiple inheritance involving them will cause only one copy of **base** to be present. Therefore, in **derived3**, there is only one copy of **base** and **ob.i = 10** is perfectly valid and unambiguous. One further point to keep in mind: Even though both **derived1** and **derived2** specify **base** as **virtual**, **base** is still present in objects of either type. For example, the following sequence is perfectly valid:

```
// define a class of type derived1
derived1 myclass;
myclass.i = 88;
```

The only difference between a normal base class and a **virtual** one is what occurs when an object inherits the base more than once. If **virtual** base classes are used, then only one base class is present in the object. Otherwise, multiple copies will be found.

# UNIT 6

# Virtual functions, Polymorphism

**6.1 Virtual function**

**6.2 Calling a Virtual function through a base class reference**

**6.3 Virtual attribute is inherited**

**6.4 Virtual functions are hierarchical**

**6.5 Pure virtual functions**

**6.6 Abstract classes Using virtual functions**

**6.7 Early and late binding.**

## 6.1 Virtual function

**Virtual Functions**

A*virtual function* is a member function that is declared within a base class and redefined by a derived class. To create a virtual function, precede the function's declaration in the base class with the keyword **virtual**. When a class containing a virtual function is inherited, the derived class redefines the virtual function to fit its own needs. In essence, virtual functions implement the "one interface, multiple methods" philosophy that underlies polymorphism.

The virtual function within the base class defines the *form* of the *interface* to that function. Each redefinition of the virtual function by a derived class implements its operation as it relates specifically to the derived class. That is, the redefinition creates a *specific method*. When accessed "normally," virtual functions behave just like any other type of class member function. However, what makes virtual functions important and capable of supporting run-time polymorphism is how they behave when accessed via a pointer. As discussed in earlier, a base-class pointer can be used to point to an object of any class derived from that base. When a base pointer points to a derived object that contains a virtual function, C++ determines which version of that function to call based upon *the type of object pointed to* by the pointer. And this determination is made *at run time*. Thus, when different objects are pointed to, different versions of the virtual function are executed. The same effect applies to base-class references. To begin, examine this short example:

```
#include <iostream>
using namespace std;
class base {
public:
virtual void vfunc() {
cout << "This is base's vfunc().\n";
}
};
class derived1 : public base {
public:
void vfunc() {
```

```
cout << "This is derived1's vfunc().\n";
}
};
class derived2 : public base {
public:
void vfunc() {
cout << "This is derived2's vfunc().\n";
}
};
int main()
{
base *p, b;
derived1 d1;
derived2 d2;
// point to base
p = &b;
p->vfunc(); // access base's vfunc()
// point to derived1
p = &d1;
p->vfunc(); // access derived1's vfunc()
// point to derived2
p = &d2;
p->vfunc(); // access derived2's vfunc()
return 0;
}
```

This program displays the following:

```
This is base's vfunc().
This is derived1's vfunc().
This is derived2's vfunc().
```

As the program illustrates, inside **base**, the virtual function **vfunc( )** is declared.

Notice that the keyword **virtual** precedes the rest of the function declaration. When **vfunc( )** is redefined by **derived1** and **derived2**, the keyword **virtual** is not needed. (However, it is not an error to include it when redefining a virtual function inside a derived class; it's just not needed.) In this program, **base** is inherited by both **derived1** and **derived2**. Inside each class definition, **vfunc( )** is redefined relative to that class. Inside **main( )**, four variables are declared:

| **Name** | **Type** |
|---|---|
| p | base class pointer |
| b | object of base |
| d1 | object of derived1 |
| d2 | object of derived2 |

Next, **p** is assigned the address of **b,** and **vfunc( )** is called via **p**. Since **p** is pointing to an object of type **base**, that version of **vfunc( )** is executed. Next, **p** is set to the address of **d1**, and again **vfunc( )** is called by using **p**. This time **p** points to an object of type **derived1**. This causes **derived1::vfunc( )** to be executed. Finally, **p** is assigned the address of **d2**, and **p >vfunc( )** causes the version of **vfunc( )** redefined inside **derived2** to be executed. The key point here is that the kind of object to which **p** points determines which version of **vfunc( )** is executed. Further, this determination is made at run time, and this process forms the basis for run-time polymorphism. Although you can call a virtual function in the "normal" manner by using an object's name and the dot operator, it is only when access is through a base-class pointer (or reference) that run-time polymorphism is achieved.

For example, assuming the preceding example, this is syntactically valid:

d2.vfunc(); // calls derived2's vfunc()

Although calling a virtual function in this manner is not wrong, it simply does not take advantage of the virtual nature of **vfunc( )**.

At first glance, the redefinition of a virtual function by a derived class appears similar to function overloading. However, this is not the case, and the term *overloading* is not applied to virtual function redefinition because several differences exist. Perhaps the most important is that the prototype for a redefined virtual function must match exactly the prototype specified in the base class. This differs from overloading a normal function, in which return types and the number and type of parameters may differ. (In fact, when you overload a function, either the number or the type of the parameters *must* differ! It is through these differences that C++ can

select the correct version of an overloaded function.) However, when a virtual function is redefined, all aspects of its prototype must be the same. If you change the prototype when you attempt to redefine a virtual function, the function will simply be considered overloaded by the C++ compiler, and its virtual nature will be lost.

Another important restriction is that virtual functions must be nonstatic members of the classes of which they are part. They cannot be **friend**s. Finally, constructor functions cannot be virtual, but destructor functions can. Because of the restrictions and differences between function overloading and virtual function redefinition, the term *overriding* is used to describe virtual function redefinition by a derived class.

## 6.2 Calling a Virtual function through a base class reference

**Calling a Virtual Function Through a Base Class Reference**

In the preceding example, a virtual function was called through a base-class pointer, but the polymorphic nature of a virtual function is also available when called through a base-class reference. As explained in Chapter 13, a reference is an implicit pointer. Thus, a base-class reference can be used to refer to an object of the base class or any object derived from that base. When a virtual function is called through a base-class reference, the version of the function executed is determined by the object being referred to at the time of the call.The most common situation in which a virtual function is invoked through a base class reference is when the reference is a function parameter.

For example, consider the following variation on the preceding program.

```
/* Here, a base class reference is used to access
a virtual function. */
#include <iostream>
using namespace std;
class base {
public:
virtual void vfunc() {
cout << "This is base's vfunc().\n";
```

```
}
};
class derived1 : public base {
public:
void vfunc() {
cout << "This is derived1's vfunc().\n";
}
};
class derived2 : public base {
public:
void vfunc() {
cout << "This is derived2's vfunc().\n";
}
};
// Use a base class reference parameter.
void f(base &r) {
r.vfunc();
}
int main()
{
base b;
derived1 d1;
derived2 d2;
f(b); // pass a base object to f()
f(d1); // pass a derived1 object to f()
f(d2); // pass a derived2 object to f()
return 0;
}
```

This program produces the same output as its preceding version. In this example, the function **f( )** defines a reference parameter of type **base**. Inside **main( )**, the function is called using objects of type **base**, **derived1**, and **derived2**. Inside **f( )**, the specific version of **vfunc( )** that is

called is determined by the type of object being referenced when the function is called. For the sake of simplicity, the rest of the examples in this chapter will call virtual functions through base-class pointers, but the effects are same for base-class references..

## 6.3 Virtual attribute is inherited

**The Virtual Attribute Is Inherited**

When a virtual function is inherited, its virtual nature is also inherited. This means that when a derived class that has inherited a virtual function is itself used as a base class for another derived class, the virtual function can still be overridden. Put differently, no matter how many times a virtual function is inherited, it remains virtual.

For example, consider this program:

```
#include <iostream>
using namespace std;
class base {
public:
virtual void vfunc() {
cout << "This is base's vfunc().\n";
}
};
class derived1 : public base {
public:
void vfunc() {
cout << "This is derived1's vfunc().\n";
}
};
/* derived2 inherits virtual function vfunc()
from derived1. */
class derived2 : public derived1 {
public:
// vfunc() is still virtual
void vfunc() {
```

```
cout << "This is derived2's vfunc().\n";
}
};
int main()
{
base *p, b;
derived1 d1;
derived2 d2;
// point to base
p = &b;
p->vfunc(); // access base's vfunc()
// point to derived1
p = &d1;
p->vfunc(); // access derived1's vfunc()
// point to derived2
p = &d2;
p->vfunc(); // access derived2's vfunc()
return 0;
}
```

As expected, the preceding program displays this output:

```
This is base's vfunc().
This is derived1's vfunc().
This is derived2's vfunc().
```

In this case, **derived2** inherits **derived1** rather than **base**, but **vfunc( )** is still virtual.

## 6.4 Virtual functions are hierarchical

**Virtual Functions Are Hierarchical**

As explained, when a function is declared as **virtual** by a base class, it may be overridden by a derived class. However, the function does not have to be overridden. When a derived class fails to override a virtual function, then when an object of that derived class accesses that function, the function defined by the base class is used.

For example, consider this program in which **derived2** does not override **vfunc( )**:

```
#include <iostream>
using namespace std;
class base {
public:
virtual void vfunc() {
cout << "This is base's vfunc().\n";
}
};
class derived1 : public base {
public:
void vfunc() {
cout << "This is derived1's vfunc().\n";
}
};
class derived2 : public base {
public:
// vfunc() not overridden by derived2, base's is used
};
int main()
{
base *p, b;
derived1 d1;
derived2 d2;
// point to base
p = &b;
p->vfunc(); // access base's vfunc()
// point to derived1
p = &d1;
p->vfunc(); // access derived1's vfunc()
// point to derived2
```

```
p = &d2;
p->vfunc(); // use base's vfunc()
return 0;
}
```

The program produces this output:

```
This is base's vfunc().
This is derived1's vfunc().
This is base's vfunc().
```

Because **derived2** does not override **vfunc( )**, the function defined by **base** is used when **vfunc( )** is referenced relative to objects of type **derived2**. The preceding program illustrates a special case of a more general rule. Because inheritance is hierarchical in C++, it makes sense that virtual functions are also hierarchical. This means that when a derived class fails to override a virtual function, the first redefinition found in reverse order of derivation is used.

For example, in the following program, **derived2** is derived from **derived1**, which is derived from **base**.However, **derived2** does not override **vfunc( )**. This means that, relative to **derived2**, the closest version of **vfunc( )** is in **derived1**. Therefore, it is **derived1::vfunc( )** that is used when an object of **derived2** attempts to call **vfunc( )**.

```
#include <iostream>
using namespace std;
class base {
public:
virtual void vfunc() {
cout << "This is base's vfunc().\n";
}
};
class derived1 : public base {
public:
void vfunc() {
cout << "This is derived1's vfunc().\n";
}
```

```
};
class derived2 : public derived1 {
public:
/* vfunc() not overridden by derived2.
In this case, since derived2 is derived from
derived1, derived1's vfunc() is used.
*/
};
int main()
{
base *p, b;
derived1 d1;
derived2 d2;
// point to base
p = &b;
p->vfunc(); // access base's vfunc()
// point to derived1
p = &d1;
p->vfunc(); // access derived1's vfunc()
// point to derived2
p = &d2;
p->vfunc(); // use derived1's vfunc()
return 0;
}
```

The program displays the following:

```
This is base's vfunc().
This is derived1's vfunc().
This is derived1's vfunc().
```

## 6.5 Pure virtual functions

**Pure Virtual Functions**

As the examples in the preceding section illustrate, when a virtual function is not redefined by a derived class, the version defined in the base class will be used. However, in many situations there can be no meaningful definition of a virtual function within a base class. For example, a base class may not be able to define an object sufficiently to allow a base-class virtual function to be created. Further, in some situations you will want to ensure that all derived classes override a virtual function. To handle these two cases, C++ supports the pure virtual function.

A *pure virtual function* is a virtual function that has no definition within the base class.

To declare a pure virtual function, use this general form:

virtual *type func-name(parameter-list*) = 0;

When a virtual function is made pure, any derived class must provide its own definition. If the derived class fails to override the pure virtual function, a compile-time error will result.

The following program contains a simple example of a pure virtual function. The base class, **number**, contains an integer called **val**, the function **setval( )**, and the pure virtual function **show( )**. The derived classes **hextype**, **dectype**, and **octtype** inherit **number** and redefine **show( )** so that it outputs the value of **val** in each respective number base (that is, hexadecimal, decimal, or octal).

```
#include <iostream>
using namespace std;
class number {
protected:
int val;
public:
void setval(int i) { val = i; }
// show() is a pure virtual function
virtual void show() = 0;
};
class hextype : public number {
public:
```

```
void show() {
cout << hex << val << "\n";
}
};
class dectype : public number {
public:
void show() {
cout << val << "\n";
}
};
class octtype : public number {
public:
void show() {
cout << oct << val << "\n";
}
};
int main()
{
dectype d;
hextype h;
octtype o;
d.setval(20);
d.show(); // displays 20 - decimal
h.setval(20);
h.show(); // displays 14 – hexadecimal
o.setval(20);
o.show(); // displays 24 - octal
return 0;
}
```

Although this example is quite simple, it illustrates how a base class may not be able to meaningfully define a virtual function. In this case, **number** simply provides the common

interface for the derived types to use. There is no reason to define **show( )** inside **number** since the base of the number is undefined. Of course, you can always create a placeholder definition of a virtual function. However, making **show( )** pure also ensures that all derived classes will indeed redefine it to meet their own needs. Keep in mind that when a virtual function is declared as pure, all derived classes must override it. If a derived class fails to do this, a compile-time error will result.

## 6.6 Abstract classes Using virtual functions

**Abstract Classes**

A class that contains at least one pure virtual function is said to be *abstract*. Because an abstract class contains one or more functions for which there is no definition (that is, a pure virtual function), no objects of an abstract class may be created. Instead, an abstract class constitutes an incomplete type that is used as a foundation for derived classes. Although you cannot create objects of an abstract class, you can create pointers and references to an abstract class. This allows abstract classes to support run-time polymorphism, which relies upon base-class pointers and references to select the proper virtual function

**Using Virtual Functions**

One of the central aspects of object-oriented programming is the principle of "one interface, multiple methods." This means that a general class of actions can be defined, the interface to which is constant, with each derivation defining its own specific operations. In concrete C++ terms, a base class can be used to define the nature of the interface to a general class. Each derived class then implements the specific operations as they relate to the type of data used by the derived type.

One of the most powerful and flexible ways to implement the "one interface, multiple methods" approach is to use virtual functions, abstract classes, and run-time polymorphism. Using these features, you create a class hierarchy that moves from general to specific (base to derived). Following this philosophy, you define all common features and interfaces in a base class. In cases where certain actions can be implemented only by the derived class, use a virtual function. In essence, in the base class you create and define everything you can that relates to the general case. The derived class fills in the specific details. Following is a simple

example that illustrates the value of the "one interface, multiple methods" philosophy. A class hierarchy is created that performs conversions from one system of units to another. (For example, liters to gallons.) The base class **convert** declares two variables, **val1** and **val2**, which hold the initial and converted values, respectively. It also defines the functions **getinit( )** and **getconv( )**, which return the initial value and the converted value. These elements of **convert** are fixed and applicable to all derived classes that will inherit **convert**. However, the function that will actually perform the conversion, **compute( )**, is a pure virtual function that must be defined by the classes derived from **convert**. The specific nature of **compute( )** will be determined by what type of conversion is taking place.

```
// Virtual function practical example.
#include <iostream>
using namespace std;
class convert {
protected:
double val1; // initial value
double val2; // converted value
public:
convert(double i) {
val1 = i;
}
double getconv() { return val2; }
double getinit() { return val1; }
virtual void compute() = 0;
};
// Liters to gallons.
class l_to_g : public convert {
public:
l_to_g(double i) : convert(i) { }
void compute() {
val2 = val1 / 3.7854;
}
```

```
};
// Fahrenheit to Celsius
class f_to_c : public convert {
public:
f_to_c(double i) : convert(i) { }
void compute() {
val2 = (val1-32) / 1.8;
}
};
int main()
{
convert *p; // pointer to base class
l_to_g lgob(4);
f_to_c fcob(70);
// use virtual function mechanism to convert
p = &lgob;
cout << p->getinit() << " liters is ";
p->compute();
cout << p->getconv() << " gallons\n"; // l_to_g
p = &fcob;
cout << p->getinit() << " in Fahrenheit is ";
p->compute();
cout << p->getconv() << " Celsius\n"; // f_to_c
return 0;
}
```

The preceding program creates two derived classes from **convert**, called **l_to_g** and **f_to_c**. These classes perform the conversions of liters to gallons and Fahrenheit to Celsius, respectively. Each derived class overrides **compute( )** in its own way to perform the desired conversion. However, even though the actual conversion (that is, method) differs between **l_to_g** and **f_to_c**, the interface remains constant. One of the benefits of derived classes and

virtual functions is that handling a new case is a very easy matter. For example, assuming the preceding program, you can add a conversion from feet to meters by including this class:

```
// Feet to meters
class f_to_m : public convert {
public:
f_to_m(double i) : convert(i) { }
void compute() {
val2 = val1 / 3.28;
}
};
```

An important use of abstract classes and virtual functions is in *class libraries*. You can create a generic, extensible class library that will be used by other programmers. Another programmer will inherit your general class, which defines the interface and all elements common to all classes derived from it, and will add those functions specific to the derived class. By creating class libraries, you are able to create and control the interface of a general class while still letting other programmers adapt it to their specific needs. One final point: The base class **convert** is an example of an abstract class. The virtual function **compute( )** is not defined within **convert** because no meaningful definition can be provided. The class **convert** simply does not contain sufficient information for **compute( )** to be defined. It is only when **convert** is inherited by a derived class that a complete type is created.

## 6.7 Early and late binding.

**Early vs. Late Binding**

Before concluding this chapter on virtual functions and run-time polymorphism, there are two terms that need to be defined because they are used frequently in discussions of C++ and object-oriented programming: *early binding* and *late binding*. *Early binding* refers to events that occur at compile time. In essence, early binding occurs when all information needed to call a function is known at compile time. (Put differently, early binding means that an object and a function call are bound during compilation.) Examples of early binding include normal function calls (including standard library functions), overloaded function calls, and overloaded operators.

The main advantage to early binding is efficiency. Because all information necessary to call a function is determined at compile time, these types of function calls are very fast. The opposite of early binding is *late binding*. As it relates to C++, late binding refers to function calls that are not resolved until run time. Virtual functions are used to achieve late binding. As you know, when access is via a base pointer or reference, the virtual function actually called is determined by the type of object pointed to by the pointer. Because in most cases this cannot be determined at compile time, the object and the function are not linked until run time. The main advantage to late binding is flexibility. Unlike early binding, late binding allows you to create programs that can respond to events occurring while the program executes without having to create a large amount of "contingency code." Keep in mind that because a function call is not resolved until run time, late binding can make for somewhat slower execution times.

# UNIT 7

## I/O System Basics, File I/0

**7.1 C++ stream classes**

**7.2 Formatted I/O**

**7.3 I/O manipulators**

**7.4 fstream and the File classes**

**7.5 File operations**

## 7.1 C++ stream classes

**The C++ Stream Classes**

As mentioned, Standard C++ provides support for its I/O system in **<iostream>**. In this header, a rather complicated set of class hierarchies is defined that supports I/O operations. The I/O classes begin with a system of template classes. As explained earlier, a template class defines the form of a class without fully specifying the data upon which it will operate. Once a template class has been defined, specific instances of it can be created. As it relates to the I/O library, Standard C++ creates two specializations of the I/O template classes: one for 8-bit characters and another for wide characters. This book will use only the 8-bit character classes since they are by far the most common. But the same techniques apply to both.

The C++ I/O system is built upon two related but different template class hierarchies. The first is derived from the low-level I/O class called **basic_streambuf**. This class supplies the basic, low-level input and output operations, and provides the underlying support for the entire C++ I/O system. Unless you are doing advanced I/O programming, you will not need to use **basic_streambuf** directly. The class hierarchy that you will most commonly be working with is derived from **basic_ios**. This is a high-level I/O class that provides formatting, error checking, and status information related to stream I/O. (A base class for **basic_ios** is called **ios_base**, which defines several nontemplate traits used by **basic_ios**.) **basic_ios** is used as a base for several derived classes, including **basic_istream**, **basic_ostream**, and **basic_iostream**. These classes are used to create streams capable of input, output, and input/output, respectively. As explained, the I/O library creates two specializations of the template class hierarchies just described: one for 8-bit characters and one for wide characters.

Here is a list of the mapping of template class names to their character and wide-character versions.

| Template Class | Characterbased Class | Wide-Characterbased Class |
|---|---|---|
| basic_streambuf | streambuf | wstreambuf |
| basic_ios | ios | wios |
| basic_istream | istream | wistream |
| basic_ostream | ostream | wostream |
| basic_iostream | iostream | wiostream |
| basic_fstream | fstream | wfstream |
| basic_ifstream | ifstream | wifstream |
| basic_ofstream | ofstream | wofstream |

The character-based names will be used throughout the remainder of this book, since they are the names that you will normally use in your programs. They are also the same names that were used by the old I/O library. This is why the old and the new I/O library are compatible at the source code level.

One last point: The **ios** class contains many member functions and variables that control or monitor the fundamental operation of a stream. It will be referred to frequently. Just remember that if you include **<iostream>** in your program, you will have access to this important class.

**C++'s Predefined Streams**

When a C++ program begins execution, four built-in streams are automatically opened.

They are:

| Stream | **Meaning** Standard | Default Device |
|---|---|---|
| cin | input Standard output | Keyboard |
| cout | Standard error output | Screen |
| cerr | Buffered version of cerr | Screen |
| clog | | Screen |

Streams **cin**, **cout**, and **cerr** correspond to C's **stdin**, **stdout**, and **stderr**.

By default, the standard streams are used to communicate with the console. However, in environments that support I/O redirection (such as DOS, Unix, OS/2, and Windows), the standard streams can be redirected to other devices or files. For the sake of simplicity, the examples in this chapter assume that no I/O redirection has occurred. Standard C++ also defines these four additional streams : **win**, **wout**, **werr**, and **wlog**. These are wide-character versions of the standard streams. Wide characters are of type **wchar_t** and are generally 16-bit quantities. Wide characters are used to hold the large character sets associated with some human languages.

## 7.2 Formatted I/O

**Formatted I/O**

The C++ I/O system allows you to format I/O operations. For example, you can set a field width, specify a number base, or determine how many digits after the decimal point will be displayed. There are two related but conceptually different ways that you can format data. First, you can directly access members of the **ios** class. Specifically, you can set various format status flags defined inside the **ios** class or call various **ios** member functions. Second, you can use special functions called *manipulators* that can be included as part of an I/O expression. We will begin the discussion of formatted I/O by using the **ios** member functions and flags.

**Formatting Using the ios Members**

Each stream has associated with it a set of format flags that control the way information is formatted. The **ios** class declares a bitmask enumeration called **fmtflags** in which the following values are defined. These values are used to set or clear the format flags. If you are using an older compiler, it may not define the **fmtflags** enumeration type. In this case, the format flags will be encoded into a long integer. When the **skipws** flag is set, leading white-space characters (spaces, tabs, and newlines) are discarded when performing input on a stream. When **skipws** is cleared, white-space characters are not discarded.When the **left** flag is set, output is left justified. When **right** is set, output is right justified. When the **internal** flag is set, a numeric value is padded to fill a field by inserting spaces between any sign or base character. If none of these flags are set, output is right justified by default.By default, numeric values are output in

decimal. However, it is possible to change the number base. Setting the **oct** flag causes output to be displayed in octal. Setting the **hex** flag causes output to be displayed in hexadecimal. To return output to decimal, set the **dec** flag.Setting **showbase** causes the base of numeric values to be shown.

For example, if the conversion base is hexadecimal, the value 1F will be displayed as 0x1F. By default, when scientific notation is displayed, the **e** is in lowercase. Also, when a hexadecimal value is displayed, the **x** is in lowercase.

When **uppercase** is set, these characters are displayed in uppercase.

Setting **showpos** causes a leading plus sign to be displayed before positive values.

Setting **showpoint** causes a decimal point and trailing zeros to be displayed for all floating-point output—whether needed or not.

By setting the **scientific** flag, floating-point numeric values are displayed using scientific notation. When **fixed** is set, floating-point values are displayed using normal notation. When neither flag is set, the compiler chooses an appropriate method. When **unitbuf** is set, the buffer is flushed after each insertion operation. When **boolalpha** is set, Booleans can be input or output using the keywords **true** and **false**. Since it is common to refer to the **oct**, **dec**, and **hex** fields, they can be collectively referred to as **basefield**. Similarly, the **left**, **right**, and **internal** fields can be referred to as **adjustfield**. Finally, the **scientific** and **fixed** fields can be referenced as **floatfield**.

## 7.3 I/O manipulators

Each stream has associated with it a set of format flags that control the way information is formatted. The **ios_base** class declares a bitmask enumeration called **fmtflags** in which the following values are defined.

adjustfield basefield boolalpha dec

fixed floatfield hex internal

left oct right scientific

showbase showpoint showpos skipws

unitbuf uppercase

These values are used to set or clear the format flags, using functions such as **setf( )** and **unsetf( )**. In addition to setting or clearing the format flags directly, you may alter the format parameters of a stream through the use of special functions called manipulators, which can be included in an I/O expression.

**Several Data Types**

In addition to the **fmtflags** type just described, the Standard C++ I/O system defines several other types.

**The streamsize and streamoff Types**

An object of type **streamsize** is capable of holding the largest number of bytes that will be transferred in any one I/O operation. It is typically some form of integer. An object of type **streamoff** is capable of holding a value that indicates an offset position within a stream. It is typically some form of integer. These types are defined in the header **<ios>**, which is automatically included by the I/O system.

**The streampos and wstreampos Types**

An object of type **streampos** is capable of holding a value that represents a position within a **char** stream. The **wstreampos** type is capable of holding a value that represents a position with a **wchar_t** stream. These are defined in **<iosfwd>**, which is automatically included by the I/O system.

**The pos_type and off_type Types**

The types **pos_type** and **off_type** create objects (typically integers) that are capable of holding a value that represents the position and an offset, respectively, within a stream. These types are defined by **ios** (and other classes) and are essentially the same as **streamoff** and **streampos** (or their wide-character equivalents).

**The openmode Type**

The type **openmode** is defined by **ios_base** and describes how a file will be opened.

**The iostate Type**

The current status of an I/O stream is described by an object of type **iostate**, which is an enumeration defined by **ios_base** that includes these members.

| Name | Meaning |
|---|---|
| goodbit | No errors occurred. |
| eofbit | End-of-file is encountered. |
| Failbit | A nonfatal I/O error has occurred. |
| Badbit | A fatal I/O error has occurred. |

**The seekdir Type**

The **seekdir** type describes how a random-access file operation will take place. It is defined within **ios_base**. Its valid values are shown here.

| | |
|---|---|
| beg | Beginning-of-file |
| cur | Current location |
| end | End-of-file |

**The failure Class**

In **ios_base** is defined the exception type **failure**. It serves as a base class for the types of exceptions that can be thrown by the I/O system. It inherits **exception** (the standard exception class). The **failure** class has the following constructor: explicit failure(const string &*str*); Here, *str* is a message that describes the error. This message can be obtained from a **failure** object by calling its **what( )** function, shown here: virtual const char *what( ) const throw( );

## 7.4 fstream and the File classes

**<fstream> and the File Classes**

To perform file I/O, you must include the header **<fstream>** in your program. It defines several classes, including **ifstream**, **ofstream**, and **fstream**. These classes are derived from **istream**, **ostream**, and **iostream**, respectively. Remember, **istream**, **ostream**, and **iostream** are derived from **ios**, so **ifstream**, **ofstream**, and **fstream** also have access to all operations defined by **ios** (discussed in the preceding chapter). Another class used by the file system is **filebuf**, which provides low-level facilities to manage a file stream. Usually, you don't use **filebuf** directly, but it is part of the other file classes.

**Opening and Closing a File**

In C++, you open a file by linking it to a stream. Before you can open a file, you must first obtain a stream. There are three types of streams: input, output, and input/output. To create an input stream, you must declare the stream to be of class **ifstream**. To create an output stream, you must declare it as class **ofstream**. Streams that will be performing both input and output operations must be declared as class **fstream**. For example, this fragment creates one input stream, one output stream, and one stream capable of both input and output:

```
ifstream in; // input
ofstream out; // output
fstream io; // input and output
```

Once you have created a stream, one way to associate it with a file is by using **open( )**. This function is a member of each of the three stream classes.

The prototype for each is

shown here:

```
void ifstream::open(const char *filename, ios::openmode mode = ios::in);
void ofstream::open(const char *filename, ios::openmode mode = ios::out | ios::trunc);
void fstream::open(const char *filename, ios::openmode mode = ios::in | ios::out);
```

Here, *filename* is the name of the file; it can include a path specifier. The value of *mode* determines how the file is opened. It must be one or more of the following values defined by **openmode**, which is an enumeration defined by **ios** (through its base class **ios_base**).

```
ios::app
ios::ate
ios::binary
ios::in
ios::out
ios::trunc
```

You can combine two or more of these values by ORing them together. Including **ios::app** causes all output to that file to be appended to the end. This value can be used only with files capable of output. Including **ios::ate** causes a seek to the end of the file to occur when the file is opened. Although **ios::ate** causes an initial seek to end-of-file, I/O operations can still occur anywhere within the file. The **ios::in** value specifies that the file is capable of input. The

**ios::out** value specifies that the file is capable of output. The **ios::binary** value causes a file to be opened in binary mode. By default, all files are opened in text mode. In text mode, various character translations may take place, such as carriage return/linefeed sequences being converted into newlines. However, when a file is opened in binary mode, no such character translations will occur. Understand that any file, whether it contains formatted text or raw data, can be opened in either binary or text mode. The only difference is whether character translations take place.

The **ios::trunc** value causes the contents of a preexisting file by the same name to be destroyed, and the file is truncated to zero length. When creating an output stream using **ofstream**, any preexisting file by that name is automatically truncated.

The following fragment opens a normal output file.

```
ofstream out;
out.open("test", ios::out);
```

However, you will seldom see **open( )** called as shown, because the *mode* parameter provides default values for each type of stream. As their prototypes show, for **ifstream**, *mode* defaults to **ios::in**; for **ofstream**, it is **ios::out | ios::trunc**; and for **fstream**, it is **ios::in | ios::out**. Therefore, the preceding statement will usually look like this:

```
out.open("test"); // defaults to output and normal file
```

*Depending on your compiler, the mode parameter for **fstream::open( )** may not default to **in | out**. Therefore, you might need to specify this explicitly.*

If **open( )** fails, the stream will evaluate to false when used in a Boolean expression. Therefore, before using a file, you should test to make sure that the open operation succeeded. You can do so by using a statement like this:

```
if(!mystream) {
cout << "Cannot open file.\n";
// handle error
}
```

Although it is entirely proper to open a file by using the **open( )** function, most of the time you will not do so because the **ifstream**, **ofstream**, and **fstream** classes have constructors that

automatically open the file. The constructors have the same parameters and defaults as the **open( )** function.

Therefore, you will most commonly see a file opened as shown here:

```
ifstream mystream("myfile"); // open file for input
```

As stated, if for some reason the file cannot be opened, the value of the associated stream variable will evaluate to false. Therefore, whether you use a constructor to open the file or an explicit call to **open( )**, you will want to confirm that the file has actually been opened by testing the value of the stream. You can also check to see if you have successfully opened a file by using the **is_open( )** function, which is a member of **fstream**, **ifstream**, and **ofstream**. It has this prototype:

```
bool is_open( );
```

It returns true if the stream is linked to an open file and false otherwise.

For example,

the following checks if **mystream** is currently open:

```
if(!mystream.is_open())
{
cout << "File is not open.\n";
// ...
```

To close a file, use the member function **close( )**. For example, to close the file linked to a stream called **mystream**, use this statement:

```
mystream.close();
```

The **close( )** function takes no parameters and returns no value.

**Reading and Writing Text Files**

It is very easy to read from or write to a text file. Simply use the **<<** and **>>** operators the same way you do when performing console I/O, except that instead of using **cin** and **cout**, substitute a stream that is linked to a file.

For example, this program creates a short inventory file that contains each item's name and its cost:

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

```
int main()
{
ofstream out("INVNTRY"); // output, normal file
if(!out) {
cout << "Cannot open INVENTORY file.\n";
return 1;
}
out << "Radios " << 39.95 << endl;
out << "Toasters " << 19.95 << endl;
out << "Mixers " << 24.80 << endl;
out.close();
return 0;
}
```

The following program reads the inventory file created by the previous program and displays its contents on the screen:

```
#include <iostream>
#include <fstream>
using namespace std;
int main()
{
ifstream in("INVNTRY"); // input
if(!in) {
cout << "Cannot open INVENTORY file.\n";
return 1;
}
char item[20];
float cost;
in >> item >> cost;
cout << item << " " << cost << "\n";
in >> item >> cost;
cout << item << " " << cost << "\n";
```

```
in >> item >> cost;
cout << item << " " << cost << "\n";
in.close();
return 0;
}
```

In a way, reading and writing files by using >> and << is like using the C-based functions **fprintf( )** and **fscanf( )**. All information is stored in the file in the same format as it would be displayed on the screen.

Following is another example of disk I/O. This program reads strings entered at the keyboard and writes them to disk. The program stops when the user enters an exclamation point.

To use the program, specify the name of the output file on the command line.

```
#include <iostream>
#include <fstream>
using namespace std;
int main(int argc, char *argv[])
{
if(argc!=2) {
cout << "Usage: output <filename>\n";
return 1;
}
ofstream out(argv[1]); // output, normal file
if(!out) {
cout << "Cannot open output file.\n";
return 1;
}
char str[80];
cout << "Write strings to disk. Enter ! to stop.\n";
do {
cout << ": ";
cin >> str;
out << str << endl;
```

```
} while (*str != '!');
out.close();
return 0;
}
```

When reading text files using the >> operator, keep in mind that certain character translations will occur. For example, white-space characters are omitted. If you want to prevent any character translations, you must open a file for binary access and use the functions discussed in the next section. When inputting, if end-of-file is encountered, the stream linked to that file will evaluate as false. (The next section illustrates this fact.)

## 7.5 File operations

**put( ) and get( )**

One way that you may read and write unformatted data is by using the member functions **get( )** and **put( )**. These functions operate on characters. That is, **get( )** will read a character and **put( )** will write a character. Of course, if you have opened the file for binary operations and are operating on a **char** (rather than a **wchar_t** stream), then these functions read and write bytes of data.

The **get( )** function has many forms, but the most commonly used version is shown here along with **put( )**:

```
istream &get(char &ch);
ostream &put(char ch);
```

The **get( )** function reads a single character from the invoking stream and puts that value in *ch*. It returns a reference to the stream. The **put( )** function writes *ch* to the stream and returns a reference to the stream.

The following program displays the contents of any file, whether it contains text or binary data, on the screen. It uses the **get( )** function.

```
#include <iostream>
#include <fstream>
using namespace std;
int main(int argc, char *argv[])
{
```

```
char ch;
if(argc!=2) {
cout << "Usage: PR <filename>\n";
return 1;
}
ifstream in(argv[1], ios::in | ios::binary);
if(!in) {
cout << "Cannot open file.";
return 1;
}
while(in) { // in will be false when eof is reached
in.get(ch);
if(in) cout << ch;
}
return 0;
}
```

As stated in the preceding section, when the end-of-file is reached, the stream associated with the file becomes false. Therefore, when **in** reaches the end of the file, it will be false, causing the **while** loop to stop.

There is actually a more compact way to code the loop that reads and displays a file, as shown here:

```
while(in.get(ch))
cout << ch;
```

This works because **get( )** returns a reference to the stream **in**, and **in** will be false when the end of the file is encountered.

The next program uses **put( )** to write all characters from zero to 255 to a file called CHARS. As you probably know, the ASCII characters occupy only about half the available values that can be held by a **char**. The other values are generally called the *extended character set* and include such things as foreign language and mathematical symbols. (Not all systems support the extended character set, but most do.)

```
#include <iostream>
```

```
#include <fstream>
using namespace std;
int main()
{
int i;
ofstream out("CHARS", ios::out | ios::binary);
if(!out) {
cout << "Cannot open output file.\n";
return 1;
}// write all characters to disk
for(i=0; i<256; i++) out.put((char) i);
out.close();
return 0;
}
```

You might find it interesting to examine the contents of the CHARS file to see what extended characters your computer has available.

**read( ) and write( )**

Another way to read and write blocks of binary data is to use C++'s **read( )** and **write( )** functions.

Their prototypes are

```
istream &read(char *buf, streamsize num);
ostream &write(const char *buf, streamsize num);
```

The **read( )** function reads *num* characters from the invoking stream and puts them in the buffer pointed to by *buf*. The **write( )** function writes *num* characters to the invoking stream from the buffer pointed to by *buf*. As mentioned in the preceding chapter, **streamsize** is a type defined by the C++ library as some form of integer. It is capable of holding the largest number of characters that can be transferred in any one I/O operation.

The next program writes a structure to disk and then reads it back in:

```
#include <iostream>
#include <fstream>
#include <cstring>
```

```
using namespace std;
struct status {
char name[80];
double balance;
unsigned long account_num;
};
int main()
{
struct status acc;
strcpy(acc.name, "Ralph Trantor");
acc.balance = 1123.23;
acc.account_num = 34235678;
// write data
ofstream outbal("balance", ios::out | ios::binary);
if(!outbal) {
cout << "Cannot open file.\n";
return 1;
}
outbal.write((char *) &acc, sizeof(struct status));
outbal.close();
// now, read back;
ifstream inbal("balance", ios::in | ios::binary);
if(!inbal) {
cout << "Cannot open file.\n";
return 1;
}
inbal.read((char *) &acc, sizeof(struct status));
cout << acc.name << endl;
cout << "Account # " << acc.account_num;
cout.precision(2);
cout.setf(ios::fixed);
```

```
cout << endl << "Balance: $" << acc.balance;
inbal.close();
return 0;
}
```

As you can see, only a single call to **read( )** or **write( )** is necessary to read or write the entire structure. Each individual field need not be read or written separately. As this example illustrates, the buffer can be any type of object.

**More get( ) Functions**

In addition to the form shown earlier, the **get( )** function is overloaded in several different ways.

The prototypes for the three most commonly used overloaded forms are shown here:

```
istream &get(char *buf, streamsize num);
istream &get(char *buf, streamsize num, char delim);
int get( );
```

The first form reads characters into the array pointed to by *buf* until either *num*-1 characters have been read, a newline is found, or the end of the file has been encountered. The array pointed to by *buf* will be null terminated by **get( )**. If the newline character is encountered in the input stream, it is *not* extracted. Instead, it remains in the stream until the next input operation.

The second form reads characters into the array pointed to by *buf* until either *num*-1 characters have been read, the character specified by *delim* has been found, or the end of the file has been encountered. The array pointed to by *buf* will be null terminated by **get( )**. If the delimiter character is encountered in the input stream, it is *not* extracted. Instead, it remains in the stream until the next input operation.

The third overloaded form of **get( )** returns the next character from the stream. It returns **EOF** if the end of the file is encountered. This form of **get( )** is similar to C's **getc( )** function.

**getline( )** Another function that performs input is **getline( )**. It is a member of each input stream class.

Its prototypes are shown here:

```
istream &getline(char *buf, streamsize num);
```

istream &getline(char *buf, streamsize num, char delim);

**Detecting EOF**

You can detect when the end of the file is reached by using the member function **eof( )**, which has this prototype:

bool eof( );

It returns true when the end of the file has been reached; otherwise it returns false.

The following program uses **eof( )** to display the contents of a file in both hexadecimal and ASCII.

```
/* Display contents of specified file in both ASCII and in hex. */
#include <iostream>
#include <fstream>
#include <cctype>
#include <iomanip>
using namespace std;
int main(int argc, char *argv[])
{
if(argc!=2) {
cout << "Usage: Display <filename>\n";
return 1;
ifstream in(argv[1], ios::in | ios::binary);
if(!in) {
cout << "Cannot open input file. \n";
return 1;
}
register int i, j;
int count = 0;
char c[16];
cout.setf(ios::uppercase);
while(!in.eof()) {
for(i=0; i<16 && !in.eof(); i++) {
in.get(c[i]);
```

```
}
if(i<16) i--; // get rid of eof
for(j=0; j<i; j++)
cout << setw(3) << hex << (int) c[j];
for(; j<16; j++) cout << " ";
cout << "\t"; for(j=0; j<i;
j++) if(isprint(c[j])) cout <<
c[j]; else cout << ".";
cout << endl;
count++;
if(count==16) {
count = 0;
cout << "Press ENTER to continue: ";
cin.get();
cout << endl;
}
}
in.close();
return 0;
}
```

**The ignore( ) Function**

You can use the **ignore( )** member function to read and discard characters from the input stream.

It has this prototype:

istream &ignore(streamsize *num*=1, int_type *delim*=EOF);

It reads and discards characters until either *num* characters have been ignored (1 by default) or the character specified by *delim* is encountered (**EOF** by default). If the delimiting character is encountered, it is not removed from the input stream. Here, **int_type** is defined as some form of integer.

The next program reads a file called TEST. It ignores characters until either a space is encountered or 10 characters have been read. It then displays the rest of the file.

```
#include <iostream>
#include <fstream>
using namespace std;
int main()
{
ifstream in("test");
if(!in) {
cout << "Cannot open file.\n";
return 1;
}
/* Ignore up to 10 characters or until first
space is found. */
in.ignore(10, ' ');
char c;
while(in) {
in.get(c);
if(in) cout << c;
}
in.close();
return 0;
}
```

**peek( ) and putback( )**

You can obtain the next character in the input stream without removing it from that stream by using **peek( ).**

It has this prototype:

int_type peek( );

It returns the next character in the stream or **EOF** if the end of the file is encountered. (**int_type** is defined as some form of integer.) You can return the last character read from a stream to that stream by using **putback( ).**

Its prototype is

istream &putback(char *c*);

where *c* is the last character read.

**flush( )**

When output is performed, data is not necessarily immediately written to the physical device linked to the stream. Instead, information is stored in an internal buffer until the buffer is full. Only then are the contents of that buffer written to disk. However, you can force the information to be physically written to disk before the buffer is full by calling **flush( )**.

Its prototype is

ostream &flush( );

Calls to **flush( )** might be warranted when a program is going to be used in adverse environments (for example, in situations where power outages occur frequently). *Closing a file or terminating a program also flushes all buffers.*

**Random Access**

In C++'s I/O system, you perform random access by using the **seekg( )** and **seekp( )** functions. Their most common forms are

istream &seekg(off_type *offset*, seekdir *origin*);

ostream &seekp(off_type *offset*, seekdir *origin*);

Here, **off_type** is an integer type defined by **ios** that is capable of containing the largest valid value that *offset* can have. **seekdir** is an enumeration defined by **ios** that determines how the seek will take place.

The C++ I/O system manages two pointers associated with a file. One is the *get pointer*, which specifies where in the file the next input operation will occur. The other is the *put pointer*, which specifies where in the file the next output operation will occur. Each time an input or output operation takes place, the appropriate pointer is automatically sequentially advanced. However, using the **seekg( )** and **seekp( )** functions allows you to access the file in a nonsequential fashion.

The **seekg**( ) function moves the associated file's current get pointer *offset* number of characters from the specified *origin*, which must be one of these three values:

ios::beg Beginning-of-file

ios::cur Current location

ios::end End-of-file

The **seekp( )** function moves the associated file's current put pointer *offset* number of characters from the specified *origin*, which must be one of the values just shown. Generally, random-access I/O should be performed only on those files opened for binary operations. The character translations that may occur on text files could cause a position request to be out of sync with the actual contents of the file.

The following program demonstrates the **seekp( )** function. It allows you to change a specific character in a file. Specify a filename on the command line, followed by the number of the character in the file you want to change, followed by the new character.

Notice that the file is opened for read/write operations.

```
#include <iostream>
#include <fstream>
#include <cstdlib>
using namespace std;
int main(int argc, char *argv[])
{
if(argc!=4) {
cout << "Usage: CHANGE <filename> <character> <char>\n";
return 1;
}
fstream out(argv[1], ios::in | ios::out | ios::binary);
if(!out) {
cout << "Cannot open file.";
return 1;
}
out.seekp(atoi(argv[2]), ios::beg);
out.put(*argv[3]);
out.close();
return 0;
}
```

For example, to use this program to change the twelfth character of a file called TEST to a Z, use this command line: change test 12 Z

The next program uses **seekg( )**. It displays the contents of a file beginning with the location you specify on the command line.

```
#include <iostream>
#include <fstream>
#include <cstdlib>
using namespace std;
int main(int argc, char *argv[])
{
char ch;
if(argc!=3) {
cout << "Usage: SHOW <filename> <starting location>\n";
return 1;
}
ifstream in(argv[1], ios::in | ios::binary);
if(!in) {
cout << "Cannot open file.";
return 1;
}
in.seekg(atoi(argv[2]), ios::beg);
while(in.get(ch))
cout << ch;
return 0;
}
```

The following program uses both **seekp( )** and **seekg( )** to reverse the first <*num*> characters in a file.

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

```
int main(int argc, char *argv[])
{
if(argc!=3) {
cout << "Usage: Reverse <filename> <num>\n";
return 1;
}
fstream inout(argv[1], ios::in | ios::out | ios::binary);
if(!inout) {
cout << "Cannot open input file.\n";
return 1;
}
long e, i, j;
char c1, c2;
e = atol(argv[2]);
for(i=0, j=e; i<j; i++, j--) {
inout.seekg(i, ios::beg);
inout.get(c1);
inout.seekg(j, ios::beg);
inout.get(c2);
inout.seekp(i, ios::beg);
inout.put(c2);
inout.seekp(j, ios::beg);
inout.put(c1);
}
inout.close();
return 0;
}
```

To use the program, specify the name of the file that you want to reverse, followed by the number of characters to reverse. For example, to reverse the first 10 characters of a file called TEST, use this command line: reverse test 10

If the file had contained this:

This is a test.

it will contain the following after the program executes:

a si sihTtest.

**Obtaining the Current File Position**

You can determine the current position of each file pointer by using these functions:

pos_type tellg( );

pos_type tellp( );

Here, **pos_type** is a type defined by **ios** that is capable of holding the largest value that either function can return. You can use the values returned by **tellg( )** and **tellp( )** as arguments to the following forms of **seekg( )** and **seekp( )**, respectively.

istream &seekg(pos_type *pos)*;

ostream &seekp(pos_type *pos*);

These functions allow you to save the current file location, perform other file operations, and then reset the file location to its previously saved location.

# UNIT 8

## Exception Handling, STL

**8.1 Exception handling fundamentals**

**8.2 Exception handling options STL: An overview**

**8.3 containers**

**8.4 vectors**

**8.5 lists**

**8.6 maps**

## 8.1 Exception handling fundamentals

**Exception Handling Fundamentals**

C++ exception handling is built upon three keywords: **try**, **catch**, and **throw**. In the most general terms, program statements that you want to monitor for exceptions are contained in a **try** block. If an exception (i.e., an error) occurs within the **try** block, it is thrown (using **throw**). The exception is caught, using **catch**, and processed. The following discussion elaborates upon this general description. Code that you want to monitor for exceptions must have been executed from within a **try** block. (Functions called from within a **try** block may also throw an exception.) Exceptions that can be thrown by the monitored code are caught by a **catch** statement, which immediately follows the **try** statement in which the exception was thrown.

The general form of **try** and **catch** are shown here.

```
try {
// try block
}
catch (type1 arg) {
// catch block
}
catch (type2 arg) {
// catch block
}
catch (type3 arg) {
// catch block
}...
catch (typeN arg) {
// catch block
}
```

The **try** can be as short as a few statements within one function or as allencompassing as enclosing the **main( )** function code within a **try** block (which effectively causes the entire program to be monitored). When an exception is thrown, it is caught by its corresponding **catch** statement, which processes the exception. There can be more than one **catch** statement

associated with a **try**. Which **catch** statement is used is determined by the type of the exception. That is, if the data type specified by a **catch** matches that of the exception, then that **Catch** statement is executed (and all others are bypassed). When an exception is caught, *arg* will receive its value. Any type of data may be caught, including classes that you create. If no exception is thrown (that is, no error occurs within the **try** block), then no **catch** statement is executed.

The general form of the **throw** statement is shown here:

throw *exception*;

**throw** generates the exception specified by *exception*. If this exception is to be caught, then **throw** must be executed either from within a **try** block itself, or from any function called from within the **try** block (directly or indirectly). If you throw an exception for which there is no applicable **catch** statement, an abnormal program termination may occur. Throwing an unhandled exception causes the standard library function **terminate( )** to be invoked. By default, **terminate( )** calls **abort( )** to stop your program, but you can specify your own termination handler, as described later in this chapter.

Here is a simple example that shows the way C++ exception handling operates.

```
// A simple exception handling example.
#include <iostream>
using namespace std;
int main()
{
cout << "Start\n";
try { // start a try block
cout << "Inside try block\n";
throw 100; // throw an error
cout << "This will not execute";
}
catch (int i) { // catch an error
cout << "Caught an exception -- value is: ";
cout << i << "\n";
}
```

```
cout << "End";
return 0;
}
```

This program displays the following output:

```
Start
Inside try block
Caught an exception -- value is: 100
End
```

Look carefully at this program. As you can see, there is a **try** block containing three statements and a **catch(int i)** statement that processes an integer exception.Within the **try** block, only two of the three statements will execute: the first **cout** statement and the **throw**. Once an exception has been thrown, control passes to the **catch** expression and the **try** block is terminated. That is, **catch** is *not* called. Rather, program execution is transferred to it. (The program's stack is automatically reset as needed to accomplish this.) Thus, the **cout** statement following the **throw** will never execute. Usually, the code within a **catch** statement attempts to remedy an error by taking appropriate action. If the error can be fixed, execution will continue with the statements following the **catch**. However, often an error cannot be fixed and a **catch** block will terminate the program with a call to **exit( )** or **abort( )**. As mentioned, the type of the exception must match the type specified in a **catch** statement. For example, in the preceding example, if you change the type in the **catch** statement to **double**, the exception will not be caught and abnormal termination will occur. This change is shown here.

```
// This example will not work.
#include <iostream>
using namespace std;
int main()
{
cout << "Start\n";
try { // start a try block
cout << "Inside try block\n";
throw 100; // throw an error
cout << "This will not execute";
```

```
}
catch (double i) { // won't work for an int exception
cout << "Caught an exception -- value is: ";
cout << i << "\n";
}
cout << "End";
return 0;
}
```

This program produces the following output because the integer exception will not be caught by the **catch(double i)** statement. (Of course, the precise message describing abnormal termination will vary from compiler to compiler.)

```
Start
Inside try block
Abnormal program termination
```

An exception can be thrown from outside the **try** block as long as it is thrown by a function that is called from within **try** block. For example, this is a valid program.

```
/* Throwing an exception from a function outside the try block. */
#include <iostream>
using namespace std;
void Xtest(int test)
{
cout << "Inside Xtest, test is: " << test << "\n";
if(test) throw test;
}
int main()
{
cout << "Start\n";
try { // start a try block
cout << "Inside try block\n";
Xtest(0);
Xtest(1);
```

```
Xtest(2);
}
catch (int i) { // catch an error
cout << "Caught an exception -- value is: ";
cout << i << "\n";
}
cout << "End";
return 0;
}
```

This program produces the following output:

```
Start
Inside try block
Inside Xtest, test is: 0
Inside Xtest, test is: 1
Caught an exception -- value is: 1
End
```

A **try** block can be localized to a function. When this is the case, each time the function is entered, the exception handling relative to that function is reset. For example, examine this program.

```
#include <iostream>
using namespace std;
// Localize a try/catch to a function.
void Xhandler(int test)
{
try{
if(test) throw test;
}
catch(int i) {
cout << "Caught Exception #: " << i << '\n';
}
}
```

```
int main()
{
cout << "Start\n";
Xhandler(1);
Xhandler(2);
Xhandler(0);
Xhandler(3);
cout << "End";
return 0;
}
```

This program displays this output:

```
Start
Caught Exception #: 1
Caught Exception #: 2
Caught Exception #: 3
End
```

As you can see, three exceptions are thrown. After each exception, the function returns. When the function is called again, the exception handling is reset. It is important to understand that the code associated with a **catch** statement will be executed only if it catches an exception. Otherwise, execution simply bypasses the **catch** altogether. (That is, execution never flows into a **catch** statement.)

For example, in the following program, no exception is thrown, so the **catch** statement does not execute.

```
#include <iostream>
using namespace std;
int main()
{
cout << "Start\n";
try { // start a try block
cout << "Inside try block\n";
cout << "Still inside try block\n";
```

```
}
catch (int i) { // catch an error
cout << "Caught an exception -- value is: ";
cout << i << "\n";
}
cout << "End";
return 0;
}
```

The preceding program produces the following output.

```
Start
Inside try block
Still inside try block
End
```

As you see, the **catch** statement is bypassed by the flow of execution.

## 8.2 Exception handling options STL: An overview

**An Overview of the STL**

Although the standard template library is large and its syntax can be intimidating, it is actually quite easy to use once you understand how it is constructed and what elements it employs. Therefore, before looking at any code examples, an overview of the STL is warranted. At the core of the standard template library are three foundational items: *containers*, *algorithms*, and *iterators*. These items work in conjunction with one another to provide off-the-shelf solutions to a variety of programming problems.

**Containers**

*Containers* are objects that hold other objects, and there are several different types.For example, the **vector** class defines a dynamic array, **deque** creates a double-ended queue, and **list** provides a linear list. These containers are called *sequence containers* because in STL terminology, a sequence is a linear list. In addition to the basic containers, the STL also defines *associative containers,* which allow efficient retrieval of values based on keys. For example, a

**map** provides access to values with unique keys. Thus, a **map** stores a key/value pair and allows a value to be retrieved given its key.

Each container class defines a set of functions that may be applied to the container. For example, a list container includes functions that insert, delete, and merge elements. A stack includes functions that push and pop values.

**Algorithms**

*Algorithms* act on containers. They provide the means by which you will manipulate the contents of containers. Their capabilities include initialization, sorting, searching, and transforming the contents of containers. Many algorithms operate on a *range* of elements within a container.

**Iterators**

*Iterators* are objects that act, more or less, like pointers. They give you the ability to cycle through the contents of a container in much the same way that you would use a pointer to cycle through an array.

**Other STL Elements**

In addition to containers, algorithms, and iterators, the STL relies upon several other standard components for support. Chief among these are allocators, predicates, comparison functions, and function objects. Each container has defined for it an *allocator*. Allocators manage memory allocation for a container. The default allocator is an object of class **allocator**, but you can define your own allocators if needed by specialized applications. For most uses, the default allocator is sufficient. Several of the algorithms and containers use a special type of function called a *predicate*.

There are two variations of predicates: unary and binary. A *unary* predicate takes one argument, while a *binary* predicate has two. These functions return true/false results. But the precise conditions that make them return true or false are defined by you. For the rest of this chapter, when a unary predicate function is required, it will be notated using the type **UnPred**. When a binary predicate is required, the type **BinPred** will be used. In a binary predicate, the arguments are always in the order of *first*, *second*. For both unary and binary predicates, the arguments will contain values of the type of objects being stored by the container. Some algorithms and classes use a special type of binary predicate that compares two elements.

Comparison functions return true if their first argument is less than their second. Comparison functions will be notated using the type **Comp**.

In addition to the headers required by the various STL classes, the C++ standard library includes the **<utility>** and **<functional>** headers, which provide support for the STL. For example, the template class **pair**, which can hold a pair of values, is defined in **<utility>**.

## 8.3 Containers

**The Container Classes**

As explained, containers are the STL objects that actually store data. The containers defined by the STL are shown. Also shown are the headers necessary to use each container. The **string** class, which manages character strings, is also a container, but it is discussed later. Since the names of the generic placeholder types in a template class declaration are arbitrary, the container classes declare **typedef**ed versions of these types. This makes the type names concrete.

Some of the most common **typedef** names are shown here:

| | |
|---|---|
| **size_type** | Some type of integer |
| **reference** | A reference to an element |
| **const_reference** | A **const** reference to an element |
| **iterator** | An iterator |
| **const_iterator** | A **const** iterator |
| **reverse_iterator** | A reverse iterator |
| **const_reverse_iterator** | A **const** reverse iterator |
| **value_type** | The type of a value stored in a container |
| **allocator_type** | The type of the allocator |
| **key_type** | The type of a key |
| **key_compare** | The type of a function that compares two keys |
| **value_compare** | The type of a function that compares two values |

## 8.4 vectors

**Vectors**

Perhaps the most general-purpose of the containers is **vector**. The **vector** class supports a dynamic array. This is an array that can grow as needed. As you know, in C++ the size of an array is fixed at compile time. While this is by far the most efficient way to implement arrays, it is also the most restrictive because the size of the array cannot be adjusted at run time to accommodate changing program conditions. A vector solves this problem by allocating memory as needed. Although a vector is dynamic, you can still use the standard array subscript notation to access its elements.

The template specification for **vector** is shown here:

template <class T, class Allocator = allocator<T> > class vector

Here, **T** is the type of data being stored and **Allocator** specifies the allocator, which defaults to the standard allocator.

**vector** has the following constructors:

explicit vector(const Allocator &*a* = Allocator( ) );

explicit vector(size_type *num*, const T &*val* = T ( ),

const Allocator &*a* = Allocator( ));

vector(const vector<T, Allocator> &*ob*);

template <class InIter> vector(InIter *start*, InIter *end*,

const Allocator &*a* = Allocator( ));

The first form constructs an empty vector. The second form constructs a vector that has *num* elements with the value *val*. The value of *val* may be allowed to default. The third form constructs a vector that contains the same elements as *ob*. The fourth form constructs a vector that contains the elements in the range specified by the iterators *start* and *end*.

For maximum flexibility and portability, any object that will be stored in a **vector** should define a default constructor. It should also define the < and == operations. Some compilers

may require that other comparison operators be defined. (Since implementations vary, consult your compiler's documentation for precise information.) All of the built-in types automatically satisfy these requirements. Although the template syntax looks rather complex, there is nothing difficult about declaring a vector.

Here are some examples:

vector<int> iv; // create zero-length int vector vector<char> cv(5); // create 5-element char vector vector<char> cv(5, 'x'); // initialize a 5-element char vector vector<int> iv2(iv); // create int vector from an int vector The following comparison operators are defined for **vector**:

==, <, <=, !=, >, >=

The subscripting operator [ ] is also defined for **vector**. This allows you to access the elements of a vector using standard array subscripting notation. Several of the member functions defined by **vectors.** (Remember, Part Four contains a complete reference to the STL classes.) Some of the most commonly used member functions are **size( )**, **begin( )**, **end( )**, **push_back( )**, **insert( )**, and **erase( )**.

The **size( )** function returns the current size of the vector. This function is quite useful because it allows you to determine the size of a vector at run time. Remember, vectors will increase in size as needed, so the size of a vector must be determined during execution, not during compilation. The **begin( )** function returns an iterator to the start of the vector. The **end( )** function returns an iterator to the end of the vector. As explained, iterators are similar to pointers, and it is through the use of the **begin( )** and **end( )** functions that you obtain an iterator to the beginning and end of a vector.

The **push_back( )** function puts a value onto the end of the vector. If necessary, the vector is increased in length to accommodate the new element. You can also add elements to the middle using **insert( )**. A vector can also be initialized. In any event, once a vector contains elements, you can use array subscripting to access or modify those elements. You can remove elements from a vector using **erase( )**.

Here is a short example that illustrates the basic operation of a vector.

```
// Demonstrate a vector.
#include <iostream>
#include <vector>
#include <cctype>
using namespace std;
int main()
{
vector<char> v(10); // create a vector of length 10
unsigned int i;
// display original size of v
cout << "Size = " << v.size() << endl;
// assign the elements of the vector some values
for(i=0; i<10; i++) v[i] = i + 'a';
// display contents of vector
cout << "Current Contents:\n";
for(i=0; i<v.size(); i++) cout << v[i] << " ";
cout << "\n\n";
cout << "Expanding vector\n";
/* put more values onto the end of the vector,
it will grow as needed */
for(i=0; i<10; i++) v.push_back(i + 10 + 'a');
// display current size of v
cout << "Size now = " << v.size() << endl;
// display contents of vector
cout << "Current contents:\n";
for(i=0; i<v.size(); i++) cout << v[i] << " ";
cout << "\n\n";
// change contents of vector
for(i=0; i<v.size(); i++) v[i] = toupper(v[i]);
cout << "Modified Contents:\n";
```

```
for(i=0; i<v.size(); i++) cout << v[i] << " ";
cout << endl;
return 0;
}
```

The output of this program is shown here:

```
Size = 10
Current Contents:
a b c d e f g h i j
Expanding vector
Size now = 20
Current contents:
a b c d e f g h i j k l m n o p q r s t
Modified Contents:
A B C D E F G H I J K L M N O P Q R S T
```

Let's look at this program carefully. In **main( )**, a character vector called **v** is created with an initial capacity of 10. That is, **v** initially contains 10 elements. This is confirmed by calling the **size( )** member function. Next, these 10 elements are initialized to the characters a through j and the contents of **v** are displayed. Notice that the standard array subscripting notation is employed. Next, 10 more elements are added to the end of **v** using the **push_back( )** function. This causes **v** to grow in order to accommodate the new elements. As the output shows, its size after these additions is 20. Finally, the values of **v**'s elements are altered using standard subscripting notation. There is one other point of interest in this program. Notice that the loops that display the contents of **v** use as their target value **v.size( )**. One of the advantages that vectors have over arrays is that it is possible to find the current size of a vector. As you can imagine, this can be quite useful in a variety of situations.

## 8.5 Lists

**Lists**

The **list** class supports a bidirectional, linear list. Unlike a vector, which supports random access, a list can be accessed sequentially only. Since lists are bidirectional, they may be accessed front to back or back to front.

A **list** has this template specification:

template <class T, class Allocator = allocator<T> > class list

Here, **T** is the type of data stored in the list. The allocator is specified by **Allocator**, which defaults to the standard allocator.

It has the following constructors:

explicit list(const Allocator &*a* = Allocator( ) );

explicit list(size_type *num*, const T &*val* = T ( ),

const Allocator &*a* = Allocator( ));

list(const list<T, Allocator> &*ob*);

template <class InIter>list(InIter *start*, InIter *end*,

const Allocator &*a* = Allocator( ));

The first form constructs an empty list. The second form constructs a list that has *num*elements with the value *val*, which can be allowed to default. The third form constructs a list that contains the same elements as *ob*. The fourth form constructs a list that contains the elements in the range specified by the iterators *start* and *end*.

The following comparison operators are defined for **list**:

==, <, <=, !=, >, >=

Some of the commonly used **list** member functions are Like vectors, elements may be put into a list by using the **push_back( )** function. You can put elements on the front of the list by using **push_front( )**. An element can also be inserted into the middle of a list by using **insert( )**. Two lists may be joined using **splice( )**. One list may be merged into another using **merge( )**. For maximum flexibility and portability, any object that will be held in a list should define a default constructor. It should also define the < operator, and possibly other comparison

operators. The precise requirements for an object that will be stored in a list vary from compiler to compiler, so you will need to check your compiler's documentation.

Here is a simple example of a **list**.

```
// List basics.
#include <iostream>
#include <list>
using namespace std;
int main()
{
list<int> lst; // create an empty list
int i;
for(i=0; i<10; i++) lst.push_back(i);
cout << "Size = " << lst.size() << endl;
cout << "Contents: ";
list<int>::iterator p = lst.begin();
while(p != lst.end()) {
cout << *p << " ";
p++;
}
cout << "\n\n";
// change contents of list
p = lst.begin();
while(p != lst.end()) {
*p = *p + 100;
p++;
}
cout << "Contents modified: ";
p = lst.begin();
while(p != lst.end()) {
cout << *p << " ";
p++;
```

```
}
return 0;
}
```

The output produced by this program is shown here:

```
Size = 10
Contents: 0 1 2 3 4 5 6 7 8 9
Contents modified: 100 101 102 103 104 105 106 107 108 109
```

This program creates a list of integers. First, an empty **list** object is created. Next, 10 integers are put into the list. This is accomplished using the **push_back( )** function, which puts each new value on the end of the existing list. Next, the size of the list and the list itself is displayed. The list is displayed via an iterator, using the following code:

```
list<int>::iterator p = lst.begin();
while(p != lst.end()) {
cout << *p << " ";
p++;
}
```

Here, the iterator **p** is initialized to point to the start of the list. Each time through the loop, **p** is incremented, causing it to point to the next element. The loop ends when **p** points to the end of the list. This code is essentially the same as was used to cycle through a vector using an iterator. Loops like this are common in STL code, and the fact that the same constructs can be used to access different types of containers is part of the power of the STL.

## 8.6 Maps

### Maps

The **map** class supports an associative container in which unique keys are mapped with values. In essence, a key is simply a name that you give to a value. Once a value has been stored, you can retrieve it by using its key. Thus, in its most general sense, a map is a list of key/value pairs. The power of a map is that you can look up a value given its key. For example, you could define a map that uses a person's name as its key and stores that person's telephone

number as its value. Associative containers are becoming more popular in programming. As mentioned, a map can hold only unique keys. Duplicate keys are not allowed.

To create a map that allows nonunique keys, use **multimap**.

The **map** container has the following template specification:

template <class Key, class T, class Comp = less<Key>,

class Allocator = allocator<pair<const key, T> > class map

Here, **Key** is the data type of the keys, **T** is the data type of the values being stored (mapped), and **Comp** is a function that compares two keys. This defaults to the standard **less( )** utility function object. **Allocator** is the allocator (which defaults to **allocator**) .

A **map** has the following constructors:

explicit map(const Comp &*cmpfn* = Comp( ),

const Allocator &*a* = Allocator( ) );

map(const map<Key, T, Comp, Allocator> &*ob*);

template <class InIter> map(InIter *start*, InIter *end*,

const Comp &*cmpfn* = Comp( ), const Allocator &*a* = Allocator( ));

The first form constructs an empty map. The second form constructs a map that contains the same elements as *ob*. The third form constructs a map that contains the elements in the range specified by the iterators *start* and *end*. The function specified by *cmpfn,* if present, determines the ordering of the map.In general, any object used as a key should define a default constructor and overload the < operator and any other necessary comparison operators. The specific requirements vary from compiler to compiler.

The following comparison operators are defined for **map**.

==, <, <=, !=, >, >=

**key_type** is the type of the key, and **value_type** represents **pair<Key, T>**.

Key/value pairs are stored in a map as objects of type **pair**, which has this template specification.

template <class Ktype, class Vtype> struct pair {

typedef Ktype first_type; // type of key

typedef Vtype second_type; // type of value

```
Ktype first; // contains the key
Vtype second; // contains the value
// constructors
pair();
pair(const Ktype &k, const Vtype &v);
template<class A, class B> pair(const<A, B> &ob);
}
```

As the comments suggest, the value in **first** contains the key and the value in **second** contains the value associated with that key. You can construct a pair using either one of **pair**'s constructors or by using **make_pair( )**, which constructs a **pair** object based upon the types of the data used as parameters. **make_pair( )** is a generic function that has this prototype.

```
template <class Ktype, class Vtype>
pair<Ktype, Vtype>
make_pair(const Ktype &k, const Vtype &v);
```

As you can see, it returns a pair object consisting of values of the types specified by *Ktype* and *Vtype*. The advantage of **make_pair( )** is that the types of the objects bei stored are determined automatically by the compiler rather than being explicitly specified by you.

The following program illustrates the basics of using a map. It stores key/value pairs that show the mapping between the uppercase letters and their ASCII character codes. Thus, the key is a character and the value is an integer. The key/value pairs stored are

A 65
B 66
C 67

and so on. Once the pairs have been stored, you are prompted for a key (i.e., a letter between A and Z), and the ASCII code for that letter is displayed.

```
// A simple map demonstration.
#include <iostream>
#include <map>
using namespace std;
```

```
int main()
{
map<char, int> m;
int i;
// put pairs into map for(i=0; i<26;
i++) { m.insert(pair<char, int>('A'+i,
65+i));
}
char ch;
cout << "Enter key: ";
cin >> ch;
map<char, int>::iterator p;
// find value given key
p = m.find(ch);
if(p != m.end())
cout << "Its ASCII value is " << p->second;
else
cout << "Key not in map.\n";
return 0;
}
```

Notice the use of the **pair** template class to construct the key/value pairs. The data types specified by **pair** must match those of the **map** into which the pairs are being inserted. Once the map has been initialized with keys and values, you can search for a value given its key by using the **find( )** function. **find( )** returns an iterator to the matching element or to the end of the map if the key is not found. When a match is found, the value associated with the key is contained in the **second** member of **pair**.

In the preceding example, key/value pairs were constructed explicitly, using **pair<char, int>**. While there is nothing wrong with this approach, it is often easier to use **make_pair( )**, which constructs a pair object based upon the types of the data used as parameters. For example, assuming the previous program, this line of code will also insert key/value pairs into **m**.

m.insert(make_pair((char)('A'+i), 65+i)); Here, the cast to **char** is needed to override the automatic conversion to **int** when **i** is added to 'A.' Otherwise, the type determination is automatic.

**Storing Class Objects in a Map**

As with all of the containers, you can use a map to store objects of types that you create. For example, the next program creates a simple phone directory. That is, it creates a map of names with their numbers. To do this, it creates two classes called **name** and **number**. Since a map maintains a sorted list of keys, the program also defines the < operator for objects of type **name**. In general, you must define the < operator for any classes that you will use as the key. (Some compilers may require that additional comparison operators be defined.)

```
// Use a map to create a phone directory.
#include <iostream>
#include <map>
#include <cstring>
using namespace std;
class name {
char str[40];
public:
name() { strcpy(str, ""); }
name(char *s) { strcpy(str, s); }
char *get() { return str; }
};
// Must define less than relative to name objects.
bool operator<(name a, name b)
{
return strcmp(a.get(), b.get()) < 0;
}
class phoneNum {
char str[80];
public:
phoneNum() { strcmp(str, ""); }
```

```
phoneNum(char *s) { strcpy(str, s); }
char *get() { return str; }
};
int main()
{
map<name, phoneNum> directory;
// put names and numbers into map
directory.insert(pair<name, phoneNum>(name("Tom"),
phoneNum("555-4533")));
directory.insert(pair<name, phoneNum>(name("Chris"),
phoneNum("555-9678")));
directory.insert(pair<name, phoneNum>(name("John"),
phoneNum("555-8195")));
directory.insert(pair<name, phoneNum>(name("Rachel"),
phoneNum("555-0809")));
// given a name, find number
char str[80];
cout << "Enter name: ";
cin >> str;
map<name, phoneNum>::iterator p;
p = directory.find(name(str));
if(p != directory.end())
cout << "Phone number: " << p->second.get();
else
cout << "Name not in directory.\n";
return 0;
}
```

Here is a sample run:

```
Enter name: Rachel
Phone number: 555-0809.
```

In the program, each entry in the map is a character array that holds a null- terminated string. Later in this chapter, you will see an easier way to write this program that uses the standard **string** type.