

1

An Overview of C++

chapter objectives

1.1 What is object-oriented programming?

1.2 Two versions of C++

1.3 C++ console I/O

1.4 C++ comments

1.5 Classes: A first look

1.6 Some differences between C and C++

1.7 Introducing function overloading

1.8 C++ keywords

C++ is an enhanced version of the C language. C++ includes everything that is part of C and adds support for object-oriented programming (OOP for short). In addition, C++ contains many improvements and features that simply make it a "better C," independent of object-oriented programming. With very few, very minor exceptions, C++ is a superset of C. While everything that you know about the C language is fully applicable to C++, understanding its enhanced features will still require a significant investment of time and effort on your part. However, the rewards of programming in C++ will more than justify the effort you put forth.

The purpose of this chapter is to introduce you to several of the most important features of C++. As you know, the elements of a computer language do not exist in a void, separate from one another. Instead, they work together to form the complete language. This interrelatedness is especially pronounced in C++. In fact, it is difficult to discuss one aspect of C++ in isolation because the features of C++ are highly integrated. To help overcome this problem, this chapter provides a brief overview of several C++ features. This overview will enable you to understand the examples discussed later in this book. Keep in mind that most topics will be more thoroughly explored in later chapters.

Since C++ was invented to support object-oriented programming, this chapter begins with a description of OOP. As you will see, many features of C++ are related to OOP in one way or another. In fact, the theory of OOP permeates C++. However, it is important to understand that C++ can be used to write programs that are and are *not* object oriented. How you use C++ is completely up to you.

At the time of this writing, the standardization of C++ is being finalized. For this reason, this chapter describes some important differences between versions of C++ that have been in common use during the past several years and the new Standard C++. Since this book teaches Standard C++, this material is especially important if you are using an older compiler.

In addition to introducing several important C++ features, this chapter also discusses some differences between C and C++ programming styles. There are several aspects of C++ that allow greater flexibility in the way that you write programs. While some of these features have little or nothing to do with object-oriented

programming, they are found in most C++ programs, so it is appropriate to discuss them early in this book.

Before you begin, a few general comments about the nature and form of C++ are in order. First, for the most part, C++ programs physically look like C programs. Like a C program, a C++ program begins execution at `main()`. To include command-line arguments, C++ uses the same `argc, argv` convention that C uses. Although C++ defines its own, object-oriented library, it also supports all the functions in the C standard library. C++ uses the same control structures as C. C++ includes all of the built-in data types defined by C.

This book assumes that you already know the C programming language. In other words, you must be able to program in C before you can learn to program in C++ by using this book. If you don't know C, a good starting place is my book *Teach Yourself C, Third Edition* (Berkeley: Osborne/McGraw-Hill, 1997). It applies the same systematic approach used in this book and thoroughly covers the entire C language.

**Note**

This book assumes that you know how to compile and execute a program using your C++ compiler. If you don't, you will need to refer to your compiler's instructions. (Because of the differences between compilers, it is impossible to give compilation instructions for each in this book.) Since programming is best learned by doing, you are strongly urged to enter, compile, and run the examples in the book in the order in which they are presented.

1.1

WHAT IS OBJECT-ORIENTED PROGRAMMING?

Object-oriented programming is a powerful way to approach the task of programming. Since its early beginnings, programming has been governed by various methodologies. At each critical point in the evolution of programming, a new approach was created to help the programmer handle increasingly complex programs. The first programs were created by toggling switches on the front panel of the computer. Obviously, this approach is suitable for only the smallest programs. Next, assembly language was invented, which allowed longer programs to be written. The next advance happened in the 1950s when the first high-level language (FORTRAN) was invented.

By using a high-level language, a programmer was able to write programs that were several thousand lines long. However, the method

of programming used early on was an ad hoc, anything-goes approach. While this is fine for relatively short programs, it yields unreadable (and unmanageable) "spaghetti code" when applied to larger programs. The elimination of spaghetti code became feasible with the invention of *structured programming languages* in the 1960s. These languages include Algol and Pascal. In loose terms, C is a structured language, and most likely the type of programming you have been doing would be called structured programming. Structured programming relies on well-defined control structures, code blocks, the absence (or at least minimal use) of the GOTO, and stand-alone subroutines that support recursion and local variables. The essence of structured programming is the reduction of a program into its constituent elements. Using structured programming, the average programmer can create and maintain programs that are up to 50,000 lines long.

Although structured programming has yielded excellent results when applied to moderately complex programs, even it fails at some point, after a program reaches a certain size. To allow more complex programs to be written, a new approach to the job of programming was needed. Towards this end, object-oriented programming was invented. OOP takes the best of the ideas embodied in structured programming and combines them with powerful new concepts that allow you to organize your programs more effectively. Object-oriented programming encourages you to decompose a problem into its constituent parts. Each component becomes a self-contained object that contains its own instructions and data that relate to that object. In this way, complexity is reduced and the programmer can manage larger programs.

All OOP languages, including C++, share three common defining traits: encapsulation, polymorphism, and inheritance. Let's look at these concepts now.

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 can 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 cannot be accessed by a piece of the program that exists outside the object. When code or data is public, other parts of your program can 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

Polymorphism (from the Greek, meaning "many forms") is the quality that allows one name to be used for two or more related but technically different purposes. As it relates to OOP, polymorphism allows one name to specify a general class of actions. Within a general class of actions, the specific action to be applied is determined by the type of data. For example, in C, which does not significantly support polymorphism, the absolute value action requires three distinct function names: **abs()**, **labs()**, and **fabs()**. These functions compute and return the absolute value of an integer, a long integer, and a floating-point value, respectively. However, in C++, which supports polymorphism, each function can be called by the same name, such as **abs()**. (One way this can be accomplished is shown later in this chapter.) The type of data used to call the function determines which specific version of the function is actually executed. As you will see, in C++, it is possible to use one function name for many different purposes. This is called *function overloading*.

More generally, the concept of polymorphism is characterized by the idea of "one interface, multiple methods," which means using a generic interface for a group of related activities. The advantage of polymorphism is that it helps to reduce complexity by allowing one interface to specify a *general class of action*. It is the compiler's job to select the *specific action* 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. As the example in the preceding paragraph illustrates, having three names for the absolute value function instead of just one makes the general activity of obtaining the absolute value of a number more complex than it actually is.

Polymorphism can be applied to operators, too. Virtually all programming languages contain a limited application of polymorphism as it relates to the arithmetic operators. For example, in C, the + sign is used to add integers, long integers, characters, and floating-point values. In these cases, the compiler automatically knows which type of arithmetic to apply. In C++, you can extend this concept to other types of data that you define. This type of polymorphism is called operator overloading.

The key point to remember about polymorphism is that it allows you to handle greater complexity by allowing the creation of standard interfaces to related activities.

INHERITANCE

Inheritance is the process by which one object can acquire the properties of another. More specifically, an object can inherit a general set of properties to which it can add those features that are specific only to itself. Inheritance is important because it allows an object to support the concept of *hierarchical classification*. Most information is made manageable by hierarchical classification. For example, think about the description of a house. A house is part of the general class called **building**. In turn, **building** is part of the more general class **structure**, which is part of the even more general class of objects that we call **man-made**. In each case, the child class inherits all those qualities associated with the parent and adds to them its own defining characteristics. Without the use of ordered classifications, each object would have to define all characteristics that relate to it explicitly. However, through inheritance, it is possible to describe an object by stating what general class (or classes) it belongs to along with those specific traits that make it unique. As you will see, inheritance plays a very important role in OOP.

EXAMPLES

1. Encapsulation is not entirely new to OOP. To a degree, encapsulation can be achieved when using the C language. For example, when you use a library function, you are, in effect, using a black-box routine, the internals of which you cannot alter or affect (except, perhaps, through malicious actions). Consider the **fopen()** function. When it is used to open a file, several internal variables are created and initialized. As far as your program is concerned, these variables are hidden and not accessible. However, C++ provides a much more secure approach to encapsulation.
 2. In the real world, examples of polymorphism are quite common. For example, consider the steering wheel on your car. It works the same whether your car uses power steering, rack-and-pinion steering, or standard, manual steering. The point is that the interface (the steering wheel) is the same no matter what type of actual steering mechanism (method) is used.
 3. Inheritance of properties and the more general concept of classification are fundamental to the way knowledge is organized. For example, celery is a member of the **vegetable** class, which is part of the **plant** class. In turn, plants are living organisms, and so forth. Without hierarchical classification, systems of knowledge would not be possible.
-

EXERCISE

1. Think about the way that classification and polymorphism play an important role in our day-to-day lives.
-

TWO VERSIONS OF C++

At the time of this writing, C++ is in the midst of a transformation. As explained in the preface to this book, C++ has been undergoing the process of standardization for the past several years. The goal has been

to create a stable, standardized, feature-rich language that will suit the needs of programmers well into the next century. As a result, there are really two versions of C++. The first is the traditional version that is based upon Bjarne Stroustrup's original designs. This is the version of C++ that has been used by programmers for the past decade. The second is the new Standard C++, which was created by Stroustrup and the ANSI/ISO standardization committee. While these two versions of C++ are very similar at their core, Standard C++ contains several enhancements not found in traditional C++. Thus, Standard C++ is essentially a superset of traditional C++.

This book teaches Standard C++. This is the version of C++ defined by the ANSI/ISO standardization committee, and it is the version implemented by all modern C++ compilers. The code in this book reflects the contemporary coding style and practices as encouraged by Standard C++. This means that what you learn in this book will be applicable today as well as tomorrow. Put directly, Standard C++ is the future. And, since Standard C++ encompasses all features found in earlier versions of C++, what you learn in this book will enable you to work in all C++ programming environments.

However, if you are using an older compiler, it might not accept all of the programs in this book. Here's why: During the process of standardization, the ANSI/ISO committee added many new features to the language. As these features were defined, they were implemented by compiler developers. Of course, there is always a lag time between the addition of a new feature to the language and the availability of the feature in commercial compilers. Since features were added to C++ over a period of years, an older compiler might not support one or more of them. This is important because two recent additions to the C++ language affect every program that you will write—even the simplest. If you are using an older compiler that does not accept these new features, don't worry. There is an easy workaround, which is described in the following paragraphs.

The differences between old-style and modern code involve two new features: new-style headers and the **namespace** statement. To demonstrate these differences we will begin by looking at two versions of a minimal, do-nothing C++ program. The first version, shown here, reflects the way C++ programs were written until recently. (That is, it uses old-style coding.)


```
/*  
    A traditional-style C++ program.  
*/  
  
#include <iostream.h>  
  
int main()  
{  
    /* program code */  
    return 0;  
}
```

Since C++ is built on C, this skeleton should be largely familiar, but pay special attention to the **#include** statement. This statement includes the file **iostream.h**, which provides support for C++'s I/O system. (It is to C++ what **stdio.h** is to C.)

Here is the second version of the skeleton, which uses the modern style:

```
/*  
    A modern-style C++ program that uses  
    the new-style headers and a namespace.  
*/  
#include <iostream>  
using namespace std;  
  
int main()  
{  
    /* program code */  
    return 0;  
}
```

Notice the two lines in this program immediately after the first comment; this is where the changes occur. First, in the **#include** statement, there is no **.h** after the name **iostream**. And second, the next line, specifying a namespace, is new. Although both the new-style headers and namespaces will be examined in detail later in this book, a brief overview is in order now.

THE NEW C++ HEADERS

As you know from your C programming experience, when you use a library function in a program, you must include its header file. This is

done using the **#include** statement. For example, in C, to include the header file for the I/O functions, you include **stdio.h** with a statement like this:

```
#include <stdio.h>
```

Here **stdio.h** is the name of the file used by the I/O functions, and the preceding statement causes that file to be included in your program. The key point is that the **#include** statement *includes a file*.

When C++ was first invented and for several years after that, it used the same style of headers as did C. In fact, Standard C++ still supports C-style headers for header files that you create and for backward compatibility. However, Standard C++ has introduced a new kind of header that is used by the Standard C++ library. The new-style headers *do not* specify filenames. Instead, they simply specify standard identifiers that might be mapped to files by the compiler, but they need not be. The new-style C++ headers are abstractions that simply guarantee that the appropriate prototypes and definitions required by the C++ library have been declared.

Since the new-style header is not a filename, it does not have a **.h** extension. Such a header consists solely of the header name contained between angle brackets. For example, here are some of the new-style headers supported by Standard C++:

```
<iostream>  
<fstream>  
<vector>  
<string>
```

The new-style headers are included using the **#include** statement. The only difference is that the new-style headers do not necessarily represent filenames.

Because C++ includes the entire C function library, it still supports the standard C-style header files associated with that library. That is, header files such as **stdio.h** and **ctype.h** are still available. However, Standard C++ also defines new-style headers that you can use in place of these header files. The C++ versions of the standard C headers simply add a **c** prefix to the filename and drop the **.h**. For example, the new-style C++ header for **math.h** is **<cmath>**, and the one for **string.h** is **<cstring>**. Although it is currently permissible to include C-style header file when using C library functions, this approach is

deprecated by Standard C++. (That is, it is not recommended.) For this reason, this book will use new-style C++ headers in all **#include** statements. If your compiler does not support new-style headers for the C function library, simply substitute the old-style, C-like headers.

Since the new-style header is a recent addition to C++, you will still find many, many older programs that don't use it. These programs instead use C-style headers, in which a filename is specified. As the old-style skeletal program shows, the traditional way to include the I/O header is as shown here:

```
#include <iostream.h>
```

This causes the file **iostream.h** to be included in your program. In general, an old-style header will use the same name as its corresponding new-style header with a **.h** appended.

As of this writing, all C++ compilers support the old-style headers. However, the old style headers have been declared obsolete, and their use in new programs is not recommended. This is why they are not used in this book.

While still common in existing C++ code, old-style headers are obsolete.



NAMESPACES

When you include a new-style header in your program, the contents of that header are contained in the **std** namespace. A *namespace* is simply a declarative region. The purpose of a namespace is to localize the names of identifiers to avoid name collisions. Traditionally, the names of library functions and other such items were simply placed into the global namespace (as they are in C). However, the contents of new-style headers are placed in the **std** namespace. We will look closely at namespaces later in this book. For now, you don't need to worry about them because you can use the statement

```
using namespace std;
```

to bring the **std** namespace into visibility (i.e., to put **std** into the global namespace). As this statement has been used, there is

no difference between working with an old-style header and a new-style one.

WORKING WITH AN OLD COMPILER

As mentioned, both namespaces and the new-style headers are recent additions to the C++ language. While virtually all new C++ compilers support these features, older compilers might not. If you have one of these older compilers, it will report one or more errors when it tries to compile the first two lines of the sample programs in this book. If this is the case, there is an easy workaround: simply use an old-style header and delete the **namespace** statement. That is, just replace

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

with

```
#include <iostream.h>
```

This change transforms a modern program into a traditional-style one. Since the old-style header reads all of its contents into the global namespace, there is no need for a **namespace** statement.

One other point: For now and for the next few years, you will see many C++ programs that use the old-style headers and that do not include a **namespace** statement. Your C++ compiler will be able to compile them just fine. For new programs, however, you should use the modern style because it is the only style of program that complies with Standard C++. While old-style programs will continue to be supported for many years, they are technically noncompliant.

EXERCISE

1. Before proceeding, try compiling the new-style skeleton program shown above. Although it does nothing, compiling it will tell you if your compiler supports the modern C++ syntax. If it does not accept the new-style headers or the **namespace** statement, substitute the old-style header as described. Remember, if your compiler does not accept new-style code, you must make this change for each program in this book.

1.3

C++ CONSOLE I/O

Since C++ is a superset of C, all elements of the C language are also contained in the C++ language. This implies that all C programs are also C++ programs by default. (Actually, there are some very minor exceptions to this rule, which are discussed later in this book.)

Therefore, it is possible to write C++ programs that look just like C programs. While there is nothing wrong with this per se, it does mean that you will not be taking full advantage of C++. To get the maximum benefit from C++, you must write C++-style programs. This means using a coding style and features that are unique to C++.

Perhaps the most common C++-specific feature used by C++ programmers is its approach to console I/O. While you may still use functions such as **printf()** and **scanf()**, C++ provides a new, and better, way to perform these types of I/O operations. In C++, I/O is performed using *I/O operators* instead of I/O functions. The output operator is **<<** and the input operator is **>>**. As you know, in C, these are the left and right shift operators, respectively. In C++, they still retain their original meanings (left and right shift) but they also take on the expanded role of performing input and output. Consider this C++ statement:

```
cout << "This string is output to the screen.\n";
```

This statement causes the string to be displayed on the computer's screen. **cout** is a predefined stream that is automatically linked to the console when a C++ program begins execution. It is similar to C's **stdout**. As in C, C++ console I/O may be redirected, but for the rest of this discussion, it is assumed that the console is being used.

By using the **<<** output operator, it is possible to output any of C++'s basic types. For example, this statement outputs the value 100.99:

```
cout << 100.99;
```

In general, to output to the console, use this form of the **<<** operator:

```
cout << expression;
```

Here *expression* can be any valid C++ expression—including another output expression.

To input a value from the keyboard, use the `>>` input operator. For example, this fragment inputs an integer value into **num**:

```
int num;
cin >> num;
```

Notice that **num** is *not* preceded by an `&`. As you know, when you use C's `scanf()` function to input values, variables must have their addresses passed to the function so they can receive the values entered by the user. This is not the case when you are using C++'s input operator. (The reason for this will become clear as you learn more about C++.)

In general, to input values from the keyboard, use this form of `>>`:

```
cin >> variable;
```

The expanded roles of `<<` and `>>` are examples of operator overloading.



Note

In order to use the C++ I/O operators, you must include the header `<iostream>` in your program. As explained earlier, this is one of C++'s standard headers and is supplied by your C++ compiler.

EXAMPLES

1. This program outputs a string, two integer values, and a double floating-point value:

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

```
int main()
{
    int i, j;
    double d;

    i = 10;
    j = 20;
    d = 99.101;
```

```
cout << "Here are some values: ";  
cout << i;  
cout << ' ';  
cout << j;  
cout << ' ';  
cout << d;  
  
return 0;  
}
```

The output of this program is shown here.

Here are some values: 10 20 99.101

**Remember**

*If you are working with an older compiler, it might not accept the new-style headers and the **namespace** statements used by this and other programs in this book. If this is the case, substitute the old-style code described in the preceding section.*

2. It is possible to output more than one value in a single I/O expression. For example, this version of the program described in Example 1 shows a more efficient way to code the I/O statements:

```
#include <iostream>  
using namespace std;  
  
int main()  
{  
    int i, j;  
    double d;  
  
    i = 10;  
    j = 20;  
    d = 99.101;  
  
    cout << "Here are some values: ";  
    cout << i << ' ' << j << ' ' << d;  
  
    return 0;  
}
```

Here the line

```
cout << i << ' ' << j << ' ' << d;
```

outputs several items in one expression. In general, you can use a single statement to output as many items as you like. If this seems confusing, simply remember that the << output operator behaves like any other C++ operator and can be part of an arbitrarily long expression.

Notice that you must explicitly include spaces between items when needed. If the spaces are left out, the data will run together when displayed on the screen.

3. This program prompts the user for an integer value:

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

int main()
{
    int i;

    cout << "Enter a value: ";
    cin >> i;
    cout << "Here's your number: " << i << "\n";

    return 0;
}
```

Here is a sample run:

```
Enter a value: 100
Here's your number: 100
```

As you can see, the value entered by the user is put into `i`.

4. The next program prompts the user for an integer value, a floating-point value, and a string. It then uses one input statement to read all three.

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

```
int main()
{
    int i;
```



```
float f;
char s[80];

cout << "Enter an integer, float, and string: ";
cin >> i >> f >> s;
cout << "Here's your data: ";
cout << i << ' ' << f << ' ' << s;

return 0;
}
```

As this example illustrates, you can input as many items as you like in one input statement. As in C, individual data items must be separated by whitespace characters (spaces, tabs, or newlines).

When a string is read, input will stop when the first whitespace character is encountered. For example, if you enter the following into the preceding program

```
10 100.12 This is a test
```

the program will display this:

```
10 100.12 This
```

The string is incomplete because the reading of the string stopped with the space after **This**. The remainder of the string is left in the input buffer, awaiting a subsequent input operation. (This is similar to inputting a string by using `scanf()` with the `%s` format.)

5. By default, when you use `>>`, all input is line buffered. This means that no information is passed to your C++ program until you press ENTER. (In C, the `scanf()` function is line buffered, so this style of input should not be new to you.) To see the effect of line-buffered input, try this program:

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

int main()
{
    char ch;

    cout << "Enter keys, x to stop.\n";
```

C++

```

do {
    cout << ": ";
    cin >> ch;
} while (ch != 'x');

return 0;
}

```

When you test this program, you will have to press ENTER after each key you type in order for the corresponding character to be sent to the program.

EXERCISES

1. Write a program that inputs the number of hours that an employee works and the employee's wage. Then display the employee's gross pay. (Be sure to prompt for input.)
2. Write a program that converts feet to inches. Prompt the user for feet and display the equivalent number of inches. Have your program repeat this process until the user enters 0 for the number of feet.
3. Here is a C program. Rewrite it so it uses C++-style I/O statements.

```

/* Convert this C program into C++ style.
   This program computes the lowest common
   denominator.
*/
#include <stdio.h>

```

```

int main(void)
{
    int a, b, d, min;

    printf("Enter two numbers: ");
    scanf("%d%d", &a, &b);
    min = a > b ? b : a;
    for(d = 2; d < min; d++)
        if(((a%d)==0) && ((b%d)==0)) break;
}

```

```
if(d==min) {
    printf("No common denominators\n");
    return 0;
}
printf("The lowest common denominator is %d\n", d);
return 0;
}
```

1.4**C++ COMMENTS**

In C++, you can include comments in your program two different ways. First, you can use the standard, C-like comment mechanism. That is, begin a comment with `/*` and end it with `*/`. As with C, this type of comment cannot be nested in C++.

The second way that you can add a remark to your C++ program is to use the *single-line comment*. A single-line comment begins with a `//` and stops at the end of the line. Other than the physical end of the line (that is, a carriage-return/linefeed combination), a single-line comment uses no comment terminator symbol.

Typically, C++ programmers use C-like comments for multiline commentaries and reserve C++-style single-line comments for short remarks.

EXAMPLES

1. Here is a program that contains both C and C++-style comments:

```
/*
   This is a C-like comment.

   This program determines whether
   an integer is odd or even.
*/

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

C++

```
int main()
{
    int num; // this is a C++ single-line comment

    // read the number
    cout << "Enter number to be tested: ";
    cin >> num;

    // see if even or odd
    if((num%2)==0) cout << "Number is even\n";
    else cout << "Number is odd\n";

    return 0;
}
```

2. While multiline comments cannot be nested, it is possible to nest a single-line comment within a multiline comment. For example, this is perfectly valid:

```
/* This is a multiline comment
   inside of which // is nested a single-line comment.
   Here is the end of the multiline comment.
*/
```

The fact that single-line comments can be nested within multiline comments makes it easier for you to "comment out" several lines of code for debugging purposes.

EXERCISES

1. As an experiment, determine whether this comment (which nests a C-like comment within a C++-style, single-line comment) is valid:

```
// This is a strange /* way to do a comment */
```

2. On your own, add comments to the answers to the exercises in Section 1.3.
-

1.5

CLASSES: A FIRST LOOK

Perhaps the single most important feature of C++ is the *class*. The class is the mechanism that is used to create objects. As such, the class is at the heart of many C++ features. Although the subject of classes is covered in great detail throughout this book, classes are so fundamental to C++ programming that a brief overview is necessary here.

A class is declared using the **class** keyword. The syntax of a **class** declaration is similar to that of a structure. Its general form is shown here:

```
class class-name {  
    // private functions and variables  
public:  
    // public functions and variables  
} object-list;
```

In a class declaration, the *object-list* is optional. As with a structure, you can declare class objects later, as needed. While the *class-name* is also technically optional, from a practical point of view it is virtually always needed. The reason for this is that the *class-name* becomes a new type name that is used to declare objects of the class.

Functions and variables declared inside a class declaration are said to be *members* of that class. By default, all functions and variables declared inside a class are private to that class. This means that they are accessible only by other members of that class. To declare public class members, the **public** keyword is used, followed by a colon. All functions and variables declared after the **public** specifier are accessible both by other members of the class and by any other part of the program that contains the class.

Here is a simple class declaration:

```
class myclass {  
    // private to myclass  
    int a;  
public:  
    void set_a(int num);  
    int get_a();
```

This class has one private variable, called **a**, and two public functions, **set_a()** and **get_a()**. Notice that functions are declared within a class using their prototype forms. Functions that are declared to be part of a class are called *member functions*.

Since **a** is private, it is not accessible by any code outside **myclass**. However, since **set_a()** and **get_a()** are members of **myclass**, they can access **a**. Further, **get_a()** and **set_a()** are declared as public members of **myclass** and can be called by any other part of the program that contains **myclass**.

Although the functions **get_a()** and **set_a()** are declared by **myclass**, they are not yet defined. To define a member function, you must link the type name of the class with the name of the function. You do this by preceding the function name with the class name followed by two colons. The two colons are called the scope resolution operator. For example, here is the way the member functions **set_a()** and **get_a()** are defined:

```
void myclass::set_a(int num)
{
    a = num;
}

int myclass::get_a()
{
    return a;
}
```

Notice that both **set_a()** and **get_a()** have access to **a**, which is private to **myclass**. Because **set_a()** and **get_a()** are members of **myclass**, they can directly access its private data.

In general, to define a member function you must use this form:

```
ret-type class-name::func-name(parameter-list)
{
    // body of function
}
```

Here *class-name* is the name of the class to which the function belongs.

The declaration of **myclass** did not define any objects of type **myclass**—it only defines the type of object that will be created when one is actually declared. To create an object, use the class name as a

type specifier. For example, this line declares two objects of type **myclass**:

```
myclass ob1, ob2; // these are objects of type myclass
```

A class declaration is a logical abstraction that defines a new type. It determines what an object of that type will look like. An object declaration creates a physical entity of that type. That is, an object occupies memory space, but a type definition does not.

Once an object of a class has been created, your program can reference its public members by using the dot (period) operator in much the same way that structure members are accessed. Assuming the preceding object declaration, the following statement calls **set_a()** for objects **ob1** and **ob2**:

```
ob1.set_a(10); // sets ob1's version of a to 10  
ob2.set_a(99); // sets ob2's version of a to 99
```

As the comments indicate, these statements set **ob1**'s copy of **a** to 10 and **ob2**'s copy to 99. Each object contains its own copy of all data declared within the class. This means that **ob1**'s **a** is distinct and different from the **a** linked to **ob2**.

Each object of a class has its own copy of every variable declared within the class.

EXAMPLES

1. As a simple first example, this program demonstrates **myclass**, described in the text. It sets the value of **a** for **ob1** and **ob2** and then displays **a**'s value for each object:

```
#include <iostream>  
using namespace std;  
  
class myclass {  
    // private to myclass  
    int a;
```

C++

```
public:
    void set_a(int num);
    int get_a();
};

void myclass::set_a(int num)
{
    a = num;
}

int myclass::get_a()
{
    return a;
}

int main()
{
    myclass ob1, ob2;

    ob1.set_a(10);
    ob2.set_a(99);

    cout << ob1.get_a() << "\n";
    cout << ob2.get_a() << "\n";
    return 0;
}
```

As you should expect, this program displays the values 10 and 99 on the screen.

2. In **myclass** from the preceding example, **a** is private. This means that only member functions of **myclass** can access it directly. (This is one reason why the public function **get_a()** is required.) If you try to access a private member of a class from some part of your program that is not a member of that class, a compile-time error will result. For example, assuming that **myclass** is defined as shown in the preceding example, the following **main()** function will cause an error:

```
// This fragment contains an error.
#include <iostream>
using namespace std;

int main()
{
```



```
myclass ob1, ob2;

ob1.a = 10; // ERROR! cannot access private member
ob2.a = 99; // by non-member functions.

cout << ob1.get_a() << "\n";
cout << ob2.get_a() << "\n";

return 0;
}
```

3. Just as there can be public member functions, there can be public member variables as well. For example, if **a** were declared in the public section of **myclass**, **a** could be referenced by any part of the program, as shown here:

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

class myclass {
public:
    // now a is public
    int a;
    // and there is no need for set_a() or get_a()
};

int main()
{
    myclass ob1, ob2;

    // here a is accessed directly
    ob1.a = 10;
    ob2.a = 99;

    cout << ob1.a << "\n";
    cout << ob2.a << "\n";

    return 0;
}
```

In this example, since **a** is declared as a public member of **myclass**, it is directly accessible from **main()**. Notice how the dot operator is used to access **a**. In general, when you are calling a member function or accessing a member variable from outside

its class, the object's name followed by the dot operator followed by the member's name is required to fully specify which object's member you are referring to.

4. To get a taste of the power of objects, let's look at a more practical example. This program creates a class called **stack** that implements a stack that can be used to store characters:

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

```
#define SIZE 10
```

```
// Declare a stack class for characters
class stack {
```

```
    char stck[SIZE]; // holds the stack
    int tos; // index of top of stack
```

```
public:
```

```
    void init(); // initialize stack
```

```
    void push(char ch); // push character on stack
```

```
    char pop(); // pop character from stack
```

```
};
```

```
// Initialize the stack
```

```
void stack::init()
```

```
{
```

```
    tos = 0;
```

```
}
```

```
// Push a character.
```

```
void stack::push(char ch)
```

```
{
```

```
    if(tos==SIZE) {
```

```
        cout << "Stack is full";
```

```
        return;
```

```
    }
```

```
    stck[tos] = ch;
```

```
    tos++;
```

```
}
```

```
// Pop a character.
```

```
char stack::pop()
```

```
{
```

```
    if(tos==0) {
```

```
    cout << "Stack is empty";
    return 0; // return null on empty stack
}
tos--;
return stck[tos];
}

int main()
{
    stack s1, s2; // create two stacks
    int i;
    // initialize the stacks
    s1.init();
    s2.init();

    s1.push('a');
    s2.push('x');
    s1.push('b');
    s2.push('y');
    s1.push('c');
    s2.push('z');

    for(i=0; i<3; i++) cout << "Pop s1: " << s1.pop() << "\n";
    for(i=0; i<3; i++) cout << "Pop s2: " << s2.pop() << "\n";

    return 0;
}
```

This program displays the following output:

```
Pop s1: c
Pop s1: b
Pop s1: a
Pop s2: z
Pop s2: y
Pop s2: x
```

Let's take a close look at this program now. The class **stack** contains two private variables: **stck** and **tos**. The array **stck** actually holds the characters pushed onto the stack, and **tos** contains the index to the top of the stack. The public stack functions are **init()**, **push()**, and **pop()**, which initialize the stack, push a value, and pop a value, respectively.

Inside `main()`, two stacks, `s1` and `s2`, are created, and three characters are pushed onto each stack. It is important to understand that each stack object is separate from the other. That is, the characters pushed onto `s1` in *no way* affect the characters pushed onto `s2`. Each object contains its own copy of `stack` and `tos`. This concept is fundamental to understanding objects. Although all objects of a class share their member functions, each object creates and maintains *its own data*.

EXERCISES

1. If you have not done so, enter and run the programs shown in the examples for this section.
 2. Create a class called `card` that maintains a library card catalog entry. Have the class store a book's title, author, and number of copies on hand. Store the title and author as strings and the number on hand as an integer. Use a public member function called `store()` to store a book's information and a public member function called `show()` to display the information. Include a short `main()` function to demonstrate the class.
 3. Create a queue class that maintains a circular queue of integers. Make the queue size 100 integers long. Include a short `main()` function that demonstrates its operation.
-

1.6

SOME DIFFERENCES BETWEEN C AND C++

Although C++ is a superset of C, there are some small differences between the two, and a few are worth knowing from the start. Before proceeding, let's take time to examine them.

First, in C, when a function takes no parameters, its prototype has the word `void` inside its function parameter list. For example, in C, if a



function called **f1()** takes no parameters (and returns a **char**), its prototype will look like this:

```
char f1(void);
```

However, in C++, the **void** is optional. Therefore, in C++, the prototype for **f1()** is usually written like this:

```
char f1();
```

C++ differs from C in the way that an empty parameter list is specified. If the preceding prototype had occurred in a C program, it would simply mean that *nothing is said about* the parameters to the function. In C++, it means that the function *has no parameters*. This is the reason that the preceding examples did not explicitly use **void** to declare an empty parameters list. (The use of **void** to declare an empty parameter list is not illegal; it is just redundant. Since most C++ programmers pursue efficiency with a nearly religious zeal, you will almost never see **void** used in this way.) Remember, in C++, these two declarations are equivalent:

```
char f1();  
char f1(void);
```

Another subtle difference between C and C++ is that in a C++ program, all functions must be prototyped. Remember, in C, prototypes are recommended but technically optional. In C++, they are required. As the examples from the previous section show, a member function's prototype contained in a class also serves as its general prototype, and no other separate prototype is required.

A third difference between C and C++ is that in C++, if a function is declared as returning a value, it must return a value. That is, if a function has a return type other than **void**, any **return** statement within that function must contain a value. In C, a non-**void** function is not required to actually return a value. If it doesn't, a garbage value is "returned."

In C, if you don't explicitly specify the return type of a function, an integer return type is assumed. C++ has dropped the "default-to-int" rule. Thus, you must explicitly declare the return type of all functions.

One other difference between C and C++ that you will commonly encounter in C++ programs has to do with where local variables can be declared. In C, local variables can be declared only at the start of a

block, prior to any "action" statements. In C++, local variables can be declared anywhere. One advantage of this approach is that local variables can be declared close to where they are first used, thus helping to prevent unwanted side effects.

Finally, C++ defines the **bool** data type, which is used to store Boolean (i.e., true/false) values. C++ also defines the keywords **true** and **false**, which are the only values that a value of type **bool** can have. In C++, the outcome of the relational and logical operators is a value of type **bool**, and all conditional statements must evaluate to a **bool** value. Although this might at first seem to be a big change from C, it isn't. In fact, it is virtually transparent. Here's why: As you know, in C, **true** is any nonzero value and **false** is 0. This still holds in C++ because any nonzero value is automatically converted into **true** and any 0 value is automatically converted into **false** when used in a Boolean expression. The reverse also occurs: **true** is converted to 1 and **false** is converted to 0 when a **bool** value is used in an integer expression. The addition of **bool** allows more thorough type checking and gives you a way to differentiate between Boolean and integer types. Of course, its use is optional; **bool** is mostly a convenience.

EXAMPLES

1. In a C program, it is common practice to declare **main()** as shown here if it takes no command-line arguments:

```
int main(void)
```

However, in C++, the use of **void** is redundant and unnecessary.

2. This short C++ program will not compile because the function **sum()** is not prototyped:

```
// This program will not compile.
```

```
#include <iostream>
```

```
using namespace std;
```

```
int main()
```

```
{
    int a, b, c;
```

```
    cout << "Enter two numbers: ";
```

```
cin >> a >> b;
c = sum(a, b);
cout << "Sum is: " << c;
```

```
return 0;
```

```
}
```

```
// This function needs a prototype.
```

```
sum(int a, int b)
```

```
{
```

```
    return a+b;
```

```
}
```

3. Here is a short program that illustrates how local variables can be declared anywhere within a block:

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

```
int main()
```

```
{
```

```
    int i; // local var declared at start of block
```

```
    cout << "Enter number: ";
```

```
    cin >> i;
```

```
    // compute factorial
```

```
    int j, fact=1; // vars declared after action statements
```

```
    for(j=i; j>=1; j--) fact = fact * j;
```

```
    cout << "Factorial is " << fact;
```

```
    return 0;
```

```
}
```

The declaration of **j** and **fact** near the point of first use is of little value in this short example; however, in large functions, the ability to declare variables close to the point of their first use can help clarify your code and prevent unintentional side effects.

4. The following program creates a Boolean variable called **outcome** and assigns it the value **false**. It then uses this variable in an **if** statement.

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

int main()
{
    bool outcome;

    outcome = false;

    if(outcome) cout << "true";
    else cout << "false";

    return 0;
}
```

As you should expect, the program displays **false**.

EXERCISES

1. The following program will not compile as a C++ program. Why not?

```
// This program has an error.
#include <iostream>
using namespace std;
```

```
int main()
```

```
{
```

```
    f();
```

```
    return 0;
```

```
}
```



```
void f()  
{  
    cout << "this won't work";  
}
```

2. On your own, try declaring local variables at various points in a C++ program. Try the same in a C program, paying attention to which declarations generate errors.

INTRODUCING FUNCTION OVERLOADING

After classes, perhaps the next most important and pervasive C++ feature is *function overloading*. Not only does function overloading provide the mechanism by which C++ achieves one type of polymorphism, it also forms the basis by which the C++ programming environment can be dynamically extended. Because of the importance of overloading, a brief introduction is given here.

In C++, two or more functions can share the same name as long as either the type of their arguments differs or the number of their arguments differs—or both. When two or more functions share the same name, they are said to be *overloaded*. Overloaded functions can help reduce the complexity of a program by allowing related operations to be referred to by the same name.

It is very easy to overload a function: simply declare and define all required versions. The compiler will automatically select the correct version based upon the number and/or type of the arguments used to call the function.

It is also possible in C++ to overload operators. However, before you can fully understand operator overloading, you will need to know more about C++.

EXAMPLES

1. One of the main uses for function overloading is to achieve compile-time polymorphism, which embodies the philosophy of one interface, many methods. As you know, in C programming, it is common to have a number of related functions that differ only by the type of data on which they operate. The classic example of this situation is found in the C standard library. As mentioned earlier in this chapter, the library contains the functions `abs()`, `labs()`, and `fabs()`, which return the absolute value of an integer, a long integer, and a floating-point value, respectively.

However, because three different names are needed due to the three different data types, the situation is more complicated than it needs to be. In all three cases, the absolute value is being returned; only the type of the data differs. In C++, you can correct this situation by overloading one name for the three types of data, as this example illustrates:

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

// Overload abs() three ways
int abs(int n);
long abs(long n);
double abs(double n);

int main()
{
    cout << "Absolute value of -10: " << abs(-10) << "\n\n";
    cout << "Absolute value of -10L: " << abs(-10L) << "\n\n";
    cout << "Absolute value of -10.01: " << abs(-10.01) << "\n\n";

    return 0;
}

// abs() for ints
int abs(int n)
{
    cout << "In integer abs()\n";
}
```

```
    return n<0 ? -n : n;
}

// abs() for longs
long abs(long n)
{
    cout << "In long abs()\n";
    return n<0 ? -n : n;
}

// abs() for doubles
double abs(double n)
{
    cout << "In double abs()\n";
    return n<0 ? -n : n;
}
```

As you can see, this program defines three functions called **abs()**—one for each data type. Inside **main()**, **abs()** is called using three different types of arguments. The compiler automatically calls the correct version of **abs()** based upon the type of data used as an argument. The program produces the following output:

```
In integer abs()
Absolute value of -10: 10

In long abs()
Absolute value of -10L: 10

In double abs()
Absolute value of -10.01: 10.01
```

Although this example is quite simple, it still illustrates the value of function overloading. Because a single name can be used to describe a general class of action, the artificial complexity caused by three slightly different names—in this case, **abs()**, **fabs()**, and **labs()**—is eliminated. You now must remember only one name—the one that describes the *general* action. It is left to the compiler to choose the appropriate *specific* version of the function (that is, the method) to call. This has the net effect of reducing complexity. Thus, through the use of polymorphism, **three** names have been reduced to one.

While the use of polymorphism in this example is fairly trivial, you should be able to see how in a very large program, the "one interface, multiple methods" approach can be quite effective.

2. Here is another example of function overloading. In this case, the function `date()` is overloaded to accept the date either as a string or as three integers. In both cases, the function displays the date passed to it.

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

void date(char *date); // date as a string
void date(int month, int day, int year); // date as numbers

int main()
{
    date("8/23/99");
    date(8, 23, 99);

    return 0;
}

// Date as string.
void date(char *date)
{
    cout << "Date: " << date << "\n";
}

// Date as integers.
void date(int month, int day, int year)
{
    cout << "Date: " << month << "/";
    cout << day << "/" << year << "\n";
}
```

This example illustrates how function overloading can provide the most natural interface to a function. Since it is very common for the date to be represented as either a string or as three integers containing the month, day, and year, you are free to select the most convenient form relative to the situation at hand.

3. So far, you have seen overloaded functions that differ in the data types of their arguments. However, overloaded functions can also differ in the number of arguments, as this example illustrates:

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

void f1(int a);
void f1(int a, int b);

int main()
{
    f1(10);
    f1(10, 20);

    return 0;
}

void f1(int a)
{
    cout << "In f1(int a)\n";
}

void f1(int a, int b)
{
    cout << "In f1(int a, int b)\n";
}
```

4. It is important to understand that the return type alone is not a sufficient difference to allow function overloading. If two functions differ only in the type of data they return, the compiler will not always be able to select the proper one to call. For example, this fragment is incorrect because it is inherently ambiguous:

```
// This is incorrect and will not compile.
int f1(int a);
double f1(int a);
```

```
f1(10); // which function does the compiler call???
```

As the comment indicates, the compiler has no way of knowing which version of `f1()` to call.

EXERCISES

1. Create a function called **sroot()** that returns the square root of its argument. Overload **sroot()** three ways: have it return the square root of an integer, a long integer, and a **double**. (To actually compute the square root, you can use the standard library function **sqrt()**.)
2. The C++ standard library contains these three functions:

```
double atof(const char *s);  
int atoi(const char *s);  
long atol(const char *s);
```

These functions return the numeric value contained in the string pointed to by **s**. Specifically, **atof()** returns a **double**, **atoi()** returns an integer, and **atol()** returns a **long**. Why is it not possible to overload these functions?

3. Create a function called **min()** that returns the smaller of the two numeric arguments used to call the function. Overload **min()** so it accepts characters, integers, and **doubles** as arguments.
4. Create a function called **sleep()** that pauses the computer for the number of seconds specified by its single argument. Overload **sleep()** so it can be called with either an integer or a string representation of an integer. For example, both of these calls to **sleep()** will cause the computer to pause for 10 seconds:

```
sleep(10);  
sleep("10");
```

Demonstrate that your functions work by including them in a short program. (Feel free to use a delay loop to pause the computer.)

1.8

C++ KEYWORDS

C++ supports all of the keywords defined by C and adds 30 of its own. The entire set of keywords defined by C++ is shown in Table 1-1. Also, early versions of C++ defined the **overload** keyword, but it is now obsolete.

SKILLS CHECK

Mastery
Skills Check

At this point you should be able to perform the following exercises and answer the questions.

1. Give brief descriptions of polymorphism, encapsulation, and inheritance.
2. How can comments be included in a C++ program?
3. Write a program that uses C++-style I/O to input two integers from the keyboard and then displays the result of raising the first to the power of the second. (For example, if the user enters 2 and 4, the result is 2^4 , or 16.)

asm	const_cast	explicit	int	register	switch	union
auto	continue	extern	long	reinterpret_cast	template	unsigned
bool	default	false	mutable	return	this	using
break	delete	float	namespace	short	throw	virtual
case	do	for	new	signed	true	void
catch	double	friend	operator	sizeof	try	volatile
char	dynamic_cast	goto	private	static	typedef	wchar_t
class	else	if	protected	static_cast	typeid	while
const	enum	inline	public	struct	typename	

TABLE 1-2 The C++ Keywords ▼

4. Create a function called `rev_str()` that reverses a string. Overload `rev_str()` so it can be called with either one character array or two. When it is called with one string, have that one string contain the reversal. When it is called with two strings, return the reversed string in the second argument. For example:

```
char s1[80], s2[80];
strcpy(s1, "hello");
rev_str(s1, s2); // reversed string goes in s2, s1 untouched
rev_str(s1); // reversed string is returned in s1
```

5. Given the following new-style C++ program, show how to change it into its old-style form.

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

```
int f(int a);
```

```
int main()
```

```
{
    cout << f(10);
```

```
    return 0;
```

```
}
```

```
int f(int a)
```

```
{
    return a * 3.1416;
}
```

6. What is the `bool` data type?

2

Introducing Classes

chapter objectives

- ✓ 2.1 Constructor and destructor functions
- ✓ 2.2 Constructors that take parameters
- ✓ 2.3 Introducing inheritance
- ✓ 2.4 Object pointers
- 2.5 Classes, structures, and unions are related
- ✓ 2.6 In-line functions
- ✓ 2.7 Automatic in-lining

THIS chapter introduces classes and objects. Several important topics are covered that relate to virtually all aspects of C++ programming, so a careful reading is advised.



Review
Skills Check

Before proceeding, you should be able to correctly answer the following questions and do the exercises.

1. Write a program that uses C++-style I/O to prompt the user for a string and then display its length.
2. Create a class that holds name and address information. Store all the information in character strings that are private members of the class. Include a public function that stores the name and address. Also include a public function that displays the name and address. (Call these functions **store()** and **display()**.)
3. Create an overloaded **rotate()** function that left-rotates the bits in its argument and returns the result. Overload it so it accepts **ints** and **longs**. (A rotate is similar to a shift except that the bit shifted off one end is shifted onto the other end.)
4. What is wrong with the following fragment?

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

```
class myclass {
    int i;
public:
    .
    .
    .
};
```

```
int main()
{
```

```
myclass ob;  
ob.i = 10;
```

2.1

CONSTRUCTOR AND DESTRUCTOR FUNCTIONS

If you have been writing programs for very long, you know that it is common for parts of your program to require initialization. The need for initialization is even more common when you are working with objects. In fact, when applied to real problems, virtually every object you create will require some sort of initialization. To address this situation, C++ allows a *constructor function* to be included in a class declaration. A class's constructor is called each time an object of that class is created. Thus, any initializations that need to be performed on an object can be done automatically by the constructor function.

A constructor function has the same name as the class of which it is a part and has no return type. For example, here is a short class that contains a constructor function:

```
#include <iostream>  
using namespace std;  
  
class myclass {  
    int a;  
public:  
    myclass(); // constructor  
    void show();  
};  
  
myclass::myclass()  
{  
    cout << "In constructor\n";  
    a = 10;  
}  
  
void myclass::show()  
{
```

44 TEACH YOURSELF

▼ C++

```
cout << a;
}

int main()
{
    myclass ob;

    ob.show();

    return 0;
}
```

In this simple example, the value of **a** is initialized by the constructor **myclass()**. The constructor is called when the object **ob** is created. An object is created when that object's declaration statement is executed. It is important to understand that in C++, a variable declaration statement is an "action statement." When you are programming in C, it is easy to think of declaration statements as simply establishing variables. However, in C++, because an object might have a constructor, a variable declaration statement may, in fact, cause a considerable number of actions to occur.

Notice how **myclass()** is defined. As stated, it has no return type. According to the C++ formal syntax rules, it is illegal for a constructor to have a return type.

For global objects, an object's constructor is called once, when the program first begins execution. For local objects, the constructor is called each time the declaration statement is executed.

The complement of a constructor is the *destructor*. This function is called when an object is destroyed. When you are working with objects, it is common to have to perform some actions when an object is destroyed. For example, an object that allocates memory when it is created will want to free that memory when it is destroyed. The name of a destructor is the name of its class, preceded by a **~**. For example, this class contains a destructor function:

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

class myclass {
    int a;
public:
```

```
myclass(); // constructor
~myclass(); // destructor
void show();
};

myclass::myclass()
{
    cout << "In constructor\n";
    a = 10;
}
myclass::~myclass()
{
    cout << "Destructing...\n";
}

void myclass::show()
{
    cout << a << "\n";
}

int main()
{
    myclass ob;

    ob.show();

    return 0;
}
```

A class's destructor is called when an object is destroyed. Local objects are destroyed when they go out of scope. Global objects are destroyed when the program ends.

It is not possible to take the address of either a constructor or a destructor.

Technically, a constructor or a destructor can perform any type of operation. The code within these functions does not have to initialize or reset anything related to the class for which they are defined. For example, a constructor for the preceding examples could have computed pi to 100 places. However, having a constructor or destructor perform actions not directly related to the initialization or orderly destruction of an object makes for very poor programming style and should be avoided.

EXAMPLES

1. You should recall that the **stack** class created in Chapter 1 required an initialization function to set the stack index variable. This is precisely the sort of operation that a constructor function was designed to perform. Here is an improved version of the **stack** class that uses a constructor to automatically initialize a stack object when it is created:

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

#define SIZE 10

// Declare a stack class for characters.
class stack {
    char stck[SIZE]; // holds the stack
    int tos; // index of top of stack
public:
    stack(); // constructor
    void push(char ch); // push character on stack
    char pop(); // pop character from stack
};

// Initialize the stack.
stack::stack()
{
    cout << "Constructing a stack\n";
    tos = 0;
}

// Push a character.
void stack::push(char ch)
{
    if(tos==SIZE) {
        cout << "Stack is full\n";
        return;
    }
    stck[tos] = ch;
    tos++;
}
```

```
// Pop a character.
char stack::pop()
{
    if(tos==0) {
        cout << "Stack is empty\n";
        return 0; // return null on empty stack
    }
    tos--;
    return stck[tos];
}

int main()
{
    // Create two stacks that are automatically initialized.
    stack s1, s2;
    int i;

    s1.push('a');
    s2.push('x');
    s1.push('b');
    s2.push('y');
    s1.push('c');
    s2.push('z');

    for(i=0; i<3; i++) cout << "Pop s1: " << s1.pop() << "\n";
    for(i=0; i<3; i++) cout << "Pop s2: " << s2.pop() << "\n";

    return 0;
}
```

As you can see, now the initialization task is performed automatically by the constructor function rather than by a separate function that must be explicitly called by the program. This is an important point. When an initialization is performed automatically when an object is created, it eliminates any prospect that, by error, the initialization will not be performed. This is another way that objects help reduce program complexity. You, as the programmer, don't need to worry about initialization—it is performed automatically when the object is brought into existence.

2. Here is an example that shows the need for both a constructor and a destructor function. It creates a simple string class, called **strtype**, that contains a string and its length. When a **strtype** object is created, memory is allocated to hold the string and its initial length is set to 0. When a **strtype** object is destroyed, that memory is released.

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

#define SIZE 255

class strtype {
    char *p;
    int len;
public:
    strtype(); // constructor
    ~strtype(); //destructor
    void set(char *ptr);
    void show();
};

// Initialize a string object.
strtype::strtype()
{
    p = (char *) malloc(SIZE);
    if(!p){
        cout << "Allocation error\n";
        exit(1);
    }
    *p = '\0';
    len = 0;
}

// Free memory when destroying string object.
strtype::~~strtype()
{
    cout << "Freeing p\n";
    free(p);
}
```



```
void strtype::set(char *ptr)
{
    if(strlen(p) >= SIZE) {
        cout << "String too big\n";
        return;
    }
    strcpy(p, ptr);
    len = strlen(p);
}

void strtype::show()
{
    cout << p << " - length: " << len;
    cout << "\n";
}


int main()
{
    strtype s1, s2;

    s1.set("This is a test.");
    s2.set("I like C++.");

    s1.show();
    s2.show();

    return 0;
}
```

This program uses **malloc()** and **free()** to allocate and free memory. While this is perfectly valid, C++ does provide another way to dynamically manage memory, as you will see later in this book.



Note

The preceding program uses the new-style headers for the C library functions used by the program. As mentioned in Chapter 1, if your compiler does not support these headers, simply substitute the standard C header files. This applies to other programs in this book in which C library functions are used.



3. Here is an interesting way to use an object's constructor and destructor. This program uses an object of the **timer** class to time the interval between when an object of type **timer** is created and when it is destroyed. When the object's destructor is called, the elapsed time is displayed. You could use an object like this to time the duration of a program or the length of time a function spends within a block. Just make sure that the object goes out of scope at the point at which you want the timing interval to end.

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

class timer {
    clock_t start;
public:
    timer(); // constructor
    ~timer(); // destructor
};

timer::timer()
{
    start = clock();
}

timer::~timer()
{
    clock_t end;

    end = clock();
    cout << "Elapsed time: " << (end-start) /
        CLOCKS_PER_SEC << "\n";
}

int main()
{
    timer ob;
    char c;

    // delay ...
```

```
cout << "Press a key followed by ENTER: ";
cin >> c;

return 0;
}
```

This program uses the standard library function `clock()`, which returns the number of clock cycles that have taken place since the program started running. Dividing this value by `CLOCKS_PER_SEC` converts the value to seconds.

EXERCISES

1. Rework the `queue` class that you developed as an exercise in Chapter 1 by replacing its initialization function with a constructor.
2. Create a class called `stopwatch` that emulates a stopwatch that keeps track of elapsed time. Use a constructor to initially set the elapsed time to 0. Provide two member functions called `start()` and `stop()` that turn on and off the timer, respectively. Include a member function called `show()` that displays the elapsed time. Also, have the destructor function automatically display elapsed time when a `stopwatch` object is destroyed. (To simplify, report the time in seconds.)
3. What is wrong with the constructor shown in the following fragment?

```
class sample {
    double a, b, c;
public:
    double sample(); // error, why?
};
```

CONSTRUCTORS THAT TAKE PARAMETERS

It is possible to pass arguments to a constructor function. To allow this, simply add the appropriate parameters to the constructor function's declaration and definition. Then, when you declare an object, specify the arguments. To see how this is accomplished, let's begin with the short example shown here:

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

class myclass {
    int a;
public:
    myclass(int x); // constructor
    void show();
};

myclass::myclass(int x)
{
    cout << "In constructor\n";
    a = x;
}

void myclass::show()
{
    cout << a << "\n";
}

int main()
{
    myclass ob(4);

    ob.show();

    return 0;
}
```

Here the constructor for **myclass** takes one parameter. The value passed to **myclass()** is used to initialize **a**. Pay special attention to how **ob** is declared in **main()**. The value 4, specified in the parentheses following **ob**, is the argument that is passed to **myclass()**'s parameter **x**, which is used to initialize **a**.

Actually, the syntax for passing an argument to a parameterized constructor is shorthand for this longer form:

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

However, most C++ programmers use the short form. Actually, there is a slight technical difference between the two forms that relates to copy constructors, which are discussed later in this book. But you don't need to worry about this distinction now.

Unlike constructor functions, destructor functions cannot have parameters. The reason for this is simple enough to understand: there exists no mechanism by which to pass arguments to an object that is being destroyed.

EXAMPLES

1. It is possible—in fact, quite common—to pass a constructor more than one argument. Here `myclass()` is passed two arguments:

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

class myclass {
    int a, b;
public:
    myclass(int x, int y); // constructor
    void show();
};

myclass::myclass(int x, int y)
{
    cout << "In constructor\n";
    a = x;
    b = y;
}

void myclass::show()
{
    cout << a << ' ' << b << "\n";
}
```

```

C++
int main()
{
    myclass ob(4, 7);

    ob.show();

    return 0;
}

```

Here 4 is passed to **x** and 7 is passed to **y**. This same general approach is used to pass any number of arguments you like (up to the limit set by the compiler, of course).

2. Here is another version of the **stack** class that uses a parameterized constructor to pass a "name" to a stack. This single-character name is used to identify the stack that is being referred to when an error occurs.

```

#include <iostream>
using namespace std;

#define SIZE 10

// Declare a stack class for characters.
class stack {
    char stck[SIZE]; // holds the stack
    int tos; // index of top of stack
    char who; // identifies stack
public:
    stack(char c); // constructor
    void push(char ch); // push character on stack
    char pop(); // pop character from stack
};

// Initialize the stack.
stack::stack(char c)
{
    tos = 0;
    who = c;
    cout << "Constructing stack " << who << "\n";
}

// Push a character.
void stack::push(char ch)
{

```

```
if(tos==SIZE) {
    cout << "Stack " << who << " is full\n";
    return;
}
stck[tos] = ch;
tos++;
}

// Pop a character.
char stack::pop()
{
    if(tos==0) {
        cout << "Stack " << who << " is empty\n";
        return 0; // return null on empty stack
    }
    tos--;
    return stck[tos];
}

int main()
{
    // Create two stacks that are automatically initialized.
    stack s1('A'), s2('B');
    int i;

    s1.push('a');
    s2.push('x');
    s1.push('b');
    s2.push('y');
    s1.push('c');
    s2.push('z');

    // This will generate some error messages.
    for(i=0; i<5; i++) cout << "Pop s1: " << s1.pop() << "\n";
    for(i=0; i<5; i++) cout << "Pop s2: " << s2.pop() << "\n";

    return 0;
}
```

Giving objects a "name," as shown in this example, is especially useful during debugging, when it is important to know which object generates an error.

3. Here is a different way to implement the **strtype** class (developed earlier) that uses a parameterized constructor function:

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

class strtype {
    char *p;
    int len;
public:
    strtype(char *ptr);
    ~strtype();
    void show();
};

strtype::strtype(char *ptr)
{
    len = strlen(ptr);
    p = (char *) malloc(len+1);
    if(!p) {
        cout << "Allocation error\n";
        exit(1);
    }
    strcpy(p, ptr);
}

strtype::~strtype()
{
    cout << "Freeing p\n";
    free(p);
}

void strtype::show()
{
    cout << p << " - length: " << len;
    cout << "\n";
}

int main()
{
    strtype s1("This is a test."), s2("I like C++.");
```



```
s1.show();
s2.show();
return 0;
}
```

In this version of **strtype**, a string is given an initial value using the constructor function.

4. Although the previous examples have used constants, you can pass an object's constructor any valid expression, including variables. For example, this program uses user input to construct an object:

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

class myclass {
    int i, j;
public:
    myclass(int a, int b);
    void show();
};

myclass::myclass(int a, int b)
{
    i = a;
    j = b;
}

void myclass::show()
{
    cout << i << ' ' << j << "\n";
}

int main()
{
    int x, y;

    cout << "Enter two integers: ";
    cin >> x >> y;

    // use variables to construct ob
    myclass ob(x, y);
}
```

```
    ob.show();  
  
    return 0;  
}
```

This program illustrates an important point about objects. They can be constructed as needed to fit the exact situation at the time of their creation. As you learn more about C++, you will see how useful constructing objects "on the fly" is.

EXERCISES

1. Change the **stack** class so it dynamically allocates memory for the stack. Have the size of the stack specified by a parameter to the constructor function. (Don't forget to free this memory with a destructor function.)
 2. Create a class called **t_and_d** that is passed the current system time and date as a parameter to its constructor when it is created. Have the class include a member function that displays this time and date on the screen. (Hint: Use the standard time and date functions found in the standard library to find and display the time and date.)
 3. Create a class called **box** whose constructor function is passed three **double** values, each of which represents the length of one side of a box. Have the **box** class compute the volume of the box and store the result in a **double** variable. Include a member function called **vol()** that displays the volume of each **box** object.
-

2.3

INTRODUCING INHERITANCE

Although inheritance is discussed more fully in Chapter 7, it needs to be introduced at this time. As it applies to C++, inheritance is the mechanism by which one class can inherit the properties of another. Inheritance allows a hierarchy of classes to be built, moving from the most general to the most specific.

To begin, it is necessary to define two terms commonly used when discussing inheritance. When one class is inherited by another, the class that is inherited is called the *base class*. The inheriting class is called the *derived class*. In general, the process of inheritance begins with the definition of a base class. The base class defines all qualities that will be common to any derived classes. In essence, the base class represents the most general description of a set of traits. A derived class inherits those general traits and adds properties that are specific to that class.

To understand how one class can inherit another, let's first begin with an example that, although simple, illustrates many key features of inheritance.

To start, here is the declaration for the base class:

```
// Define base class.
class B {
    int i;
public:
    void set_i(int n);
    int get_i();
};
```

Using this base class, here is a derived class that inherits it:

```
// Define derived class.
class D : public B {
    int j;
public:
    void set_j(int n);
    int mul();
};
```

Look closely at this declaration. Notice that after the class name **D** there is a colon followed by the keyword **public** and the class name **B**. This tells the compiler that class **D** will inherit all components of class **B**. The keyword **public** tells the compiler that **B** will be inherited such that all public elements of the base class will also be public elements of the derived class. However, all private elements of the base class remain private to it and are not directly accessible by the derived class.

Here is an entire program that uses the **B** and **D** classes:

```
// A simple example of inheritance.
```

```
#include <iostream>
```

```
using namespace std;
```

```
// Define base class.
```

```
class B {
```

```
    int i;
```

```
public:
```

```
    void set_i(int n);
```

```
    int get_i();
```

```
};
```

```
// Define derived class.
```

```
class D : public B {
```

```
    int j;
```

```
public:
```

```
    void set_j(int n);
```

```
    int mul();
```

```
};
```

```
// Set value i in base.
```

```
void B::set_i(int n)
```

```
{
```

```
    i = n;
```

```
}
```

```
// Return value of i in base.
```

```
int B::get_i()
```

```
{
```

```
    return i;
```

```
}
```

```
// Set value of j in derived.
```

```

void D::set_j(int, n)
{
    j = n;
}

// Return value of base's i times derived's j.
int D::mul()
{
    // derived class can call base class public member functions
    return j * get_i();
}

int main()
{
    D ob;

    ob.set_i(10); // load i in base
    ob.set_j(4);  // load j in derived

    cout << ob.mul(); // displays 40

    return 0;
}

```

Look at the definition of `mul()`. Notice that it calls `get_i()`, which is a member of the base class **B**, not of **D**, without linking it to any specific object. This is possible because the public members of **B** become public members of **D**. However, the reason that `mul()` must call `get_i()` instead of accessing `i` directly is that the private members of a base class (in this case, `i`) remain private to it and not accessible by any derived class. The reason that private members of a class are not accessible to derived classes is to maintain encapsulation. If the private members of a class could be made public simply by inheriting the class, encapsulation could be easily circumvented.

The general form used to inherit a base class is shown here:

```
class derived-class-name : access-specifier base-class-name {
```

```
};
```

Here *access-specifier* is one of the following three keywords: **public**, **private**, or **protected**. For now, just use **public** when inheriting a class. A complete description of the access specifiers will be given later in this book.

EXAMPLE

1. Here is a program that defines a generic base class called **fruit** that describes certain characteristics of fruit. This class is inherited by two derived classes called **Apple** and **Orange**. These classes supply specific information to **fruit** that are related to these types of fruit.

```
// An example of class inheritance.
#include <iostream>
#include <cstring>
using namespace std;

enum yn {no, yes};
enum color {red, yellow, green, orange};

void out(enum yn x);

char *c[] = {
    "red", "yellow", "green", "orange"};

// Generic fruit class.
class fruit {
// in this base, all elements are public
public:
    enum yn annual;
    enum yn perennial;
    enum yn tree;
    enum yn tropical;
    enum color clr;
    char name[40];
};

// Derive Apple class.
class Apple : public fruit {
    enum yn cooking;
    enum yn crunchy;
```

```
enum yn eating;
public:
    void seta(char *n, enum color c, enum yn ck, enum yn crchy,
              enum yn e);
    void show();
};
```

// Derive orange class.

```
class Orange : public fruit {
    enum yn juice;
    enum yn sour;
    enum yn eating;
public:
    void seto(char *n, enum color c, enum yn j, enum yn sr,
              enum yn e);
    void show();
};
```

```
void Apple::seta(char *n, enum color c, enum yn ck,
                 enum yn crchy, enum yn e)
```

```
{
    strcpy(name, n);
    annual = no;
    perennial = yes;

    tree = yes;
    tropical = no;
    clr = c;
    cooking = ck;
    crunchy = crchy;
    eating = e;
}
```

```
void Orange::seto(char *n, enum color c, enum yn j,
                  enum yn sr, enum yn e)
```

```
{
    strcpy(name, n);
    annual = no;
    perennial = yes;
    tree = yes;
    tropical = yes;
    clr = c;
    juice = j;
    sour = sr;
}
```

```
        eating = e;
    }

void Apple::show()
{
    cout << name << " apple is: " << "\n";
    cout << "Annual: "; out(annual);
    cout << "Perennial: "; out(perennial);
    cout << "Tree: "; out(tree);
    cout << "Tropical: "; out(tropical);
    cout << "Color: " << c[clr] << "\n";
    cout << "Good for cooking: "; out(cooking);
    cout << "Crunchy: "; out(crunchy);
    cout << "Good for eating: "; out(eating);
    cout << "\n";
}

void Orange::show()
{
    cout << name << " orange is: " << "\n";
    cout << "Annual: "; out(annual);
    cout << "Perennial: "; out(perennial);
    cout << "Tree: "; out(tree);
    cout << "Tropical: "; out(tropical);
    cout << "Color: " << c[clr] << "\n";
    cout << "Good for juice: "; out(juice);
    cout << "Sour: "; out(sour);
    cout << "Good for eating: "; out(eating);
    cout << "\n";
}

void out(enum yn x)
{
    if(x==no) cout << "no\n";
    else cout << "yes\n";
}

int main()
{
    Apple a1, a2;
    Orange o1, o2;

    a1.seta("Red Delicious", red, no, yes, yes);
```



```

a2.seta("Jonathan", red, yes, no, yes);

o1.seto("Navel", orange, no, no, yes);
o2.seto("Valencia", orange, yes, yes, no);

a1.show();
a2.show();

o1.show();
o2.show();

return 0;
}

```

As you can see, the base class **fruit** defines several characteristics that are common to all types of fruit. (Of course, in order to keep this example short enough to fit conveniently in a book, the **fruit** class is somewhat simplified.) For example, all fruit grows on either annual or perennial plants. All fruit grows either on trees or on other types of plants, such as vines or bushes. All fruit has a color and a name. This base class is then inherited by the **Apple** and **Orange** classes. Each of these classes supplies information specific to its type of fruit.

This example illustrates the basic reason for inheritance. Here, a base class is created that defines the general traits associated with *all* fruit. It is left to the derived classes to supply those traits that are specific to each *individual* case.

This program illustrates another important fact about inheritance: A base class is not exclusively "owned" by a derived class. A base class can be inherited by any number of classes.

EXERCISE

1. Given the following base class,

```

class area_cl {
public:
    double height;

```

```
double width;
};
```

create two derived classes called **rectangle** and **isosceles** that inherit **area_cl**. Have each class include a function called **area()** that returns the area of a rectangle or isosceles triangle, as appropriate. Use parameterized constructors to initialize **height** and **width**.

2.4

OBJECT POINTERS

So far, you have been accessing members of an object by using the dot operator. This is the correct method when you are working with an object. However, it is also possible to access a member of an object via a pointer to that object. When a pointer is used, the arrow operator ($->$) rather than the dot operator is employed. (This is exactly the same way the arrow operator is used when given a pointer to a structure.)

You declare an **object pointer** just like you declare a pointer to any other type of variable. Specify its class name, and then precede the variable name with an asterisk. To obtain the address of an object, precede the object with the **&** operator, just as you do when taking the address of any other type of variable.

Just like pointers to other types, an object pointer, when incremented, will point to the next object of its type.

EXAMPLE

1. Here is a simple example that uses an object pointer:

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

class myclass {
    int a;
public:
    myclass(int x); // constructor
```

```
int get();
};

myclass::myclass(int x)
{
    a = x;
}

int myclass::get()
{
    return a;
}

int main()
{
    myclass ob(120); // create object
    myclass *p; // create pointer to object

    p = &ob; // put address of ob into p

    cout << "Value using object: " << ob.get();
    cout << "\n";

    cout << "Value using pointer: " << p->get();

    return 0;
}
```

Notice how the declaration

```
myclass *p;
```

creates a pointer to an object of **myclass**. It is important to understand that creation of an object pointer does *not* create an object—it creates just a pointer to one. The address of **ob** is put into **p** by using this statement:

```
p = &ob;
```

Finally, the program shows how the members of an object can be accessed through a pointer.

We will come back to the subject of object pointers in Chapter 4, once you know more about C++. There are several special features that relate to them.

CLASSES, STRUCTURES, AND UNIONS ARE RELATED

As you have seen, the class is syntactically similar to the structure. You might be surprised to learn, however, that the class and the structure have virtually identical capabilities. In C++, the definition of a structure has been expanded so that it can also include member functions, including constructor and destructor functions, in just the same way that a class can. In fact, the only difference between a structure and a class is that, by default, the members of a class are private but the members of a structure are public. The expanded syntax of a structure is shown here:

```
struct type-name {
    // public function and data members
private:
    // private function and data members
} object-list;
```

In fact, according to the formal C++ syntax, both **struct** and **class** create new class types. Notice that a new keyword is introduced. It is **private**, and it tells the compiler that the members that follow are private to that class.

On the surface, there is a seeming redundancy in the fact that structures and classes have virtually identical capabilities. Many newcomers to C++ wonder why this apparent duplication exists. In fact, it is not uncommon to hear the suggestion that the **class** keyword is unnecessary.

The answer to this line of reasoning has both a "strong" and "weak" form. The "strong" (or compelling) reason concerns maintaining upward compatibility from C. In C++, a C-style structure is also perfectly acceptable in a C++ program. Since in C all structure members are public by default, this convention is also maintained in C++. Further, because **class** is a syntactically separate entity from **struct**, the definition of a class is free to evolve in a way that will not be compatible with a C-like structure definition. Since the two are separated, the future direction of C++ is not restricted by compatibility concerns.

The "weak" reason for having two similar constructs is that there is no disadvantage to expanding the definition of a structure in C++ to include member functions.

Although structures have the same capabilities as classes, most programmers restrict their use of structures to adhere to their C-like form and do not use them to include function members. Most programmers use the **class** keyword when defining objects that contain both data and code. However, this is a stylistic matter and is subject to your own preference. (After this section, this book reserves the use of **struct** for objects that have no function members.)

If you find the connection between classes and structures interesting, so will you find this next revelation about C++: unions and classes are also related. In C++, a union defines a class type that can contain both functions and data as members. A union is like a structure in that, by default, all members are public until the **private** specifier is used. In a union, however, all data members share the same memory location (just as in C). Unions can contain constructor and destructor functions. Fortunately, C unions are upwardly compatible with C++ unions.

Although structures and classes seem on the surface to be redundant, this is not the case with unions. In an object-oriented language, it is important to preserve encapsulation. Thus, the union's ability to link code and data allows you to create class types in which all data uses a shared location. This is something that you cannot do using a class.

There are several restrictions that apply to unions as they relate to C++. First, they cannot inherit any other class and they cannot be used as a base class for any other type. Unions must not have any **static** members. They also must not contain any object that has a constructor or destructor. The union, itself, *can* have a constructor and destructor, though. Finally, unions cannot have virtual member functions. (Virtual functions are described later in this book.)

There is a special type of union in C++ called an *anonymous union*. An anonymous union does not have a type name, and no variables can be declared for this sort of union. Instead, an anonymous union tells the compiler that its members will share the same memory location. However, in all other respects, the members act and are treated like normal variables. That is, the members are accessed directly, without the dot operator syntax. For example, examine this fragment:

```
union { // an anonymous union
    int i;
    char ch[4];
} ;
```

```
i = 10; // access i and ch directly
ch[0] = 'X';
```

As you can see, **i** and **ch** are accessed directly because they are not part of any object. They do, however, share the same memory space.

The reason for the anonymous union is that it gives you a simple way to tell the compiler that you want two or more variables to share the same memory location. Aside from this special attribute, members of an anonymous union behave like other variables.

Anonymous unions have all of the restrictions that apply to normal unions, plus these additions. A global anonymous union must be declared **static**. An anonymous union cannot contain private members. The names of the members of an anonymous union must not conflict with other identifiers within the same scope.

EXAMPLES

1. Here is a short program that uses **struct** to create a class:

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

// use struct to define a class type
struct st_type {
    st_type(double b, char *n);
    void show();
private:
    double balance;
    char name[40];
};

st_type::st_type(double b, char *n)
{
    balance = b;
    strcpy(name, n);
}

void st_type::show()
{
    cout << "Name: " << name;
    cout << ": $" << balance;
    if(balance<0.0) cout << "****";
    cout << "\n";
}
```

```
}  
  
int main()  
{  
    st_type acc1(100.12, "Johnson");  
    st_type acc2(-12.34, "Hedricks");  
  
    acc1.show();  
    acc2.show();  
    return 0;  
}
```

Notice that, as stated, the members of a structure are public by default. The **private** keyword must be used to declare private members.

Also, notice one difference between C-like structures and C++-like structures. In C++, the structure tag-name also becomes a complete type name that can be used to declare objects. In C, the tag-name requires that the keyword **struct** precede it before it becomes a complete type.

Here is the same program, rewritten using a class:

```
#include <iostream>  
#include <cstring>  
using namespace std;  
  
class cl_type {  
    double balance;  
    char name[40];  
public:  
    cl_type(double b, char *n);  
    void show();  
};  
  
cl_type::cl_type(double b, char *n)  
{  
    balance = b;  
    strcpy(name, n);  
}  
  
void cl_type::show()  
{  
    cout << "Name: " << name;  
    cout << ": $" << balance;
```

```

    if(balance<0.0) cout << "****";
    cout << "\n";
}

```

```

int main()
{
    cl_type acc1(100.12, "Johnson");
    cl_type acc2(-12.34, "Hedricks");

    acc1.show();
    acc2.show();

    return 0;
}

```

2. Here is an example that uses a union to display the binary bit pattern, byte by byte, contained within a **double** value.

```

#include <iostream>
using namespace std;

union bits {
    bits(double n);
    void show_bits();
    double d;
    unsigned char c[sizeof(double)];
};

bits::bits(double n)
{
    d = n;
}

void bits::show_bits()
{
    int i, j;

    for(j = sizeof(double)-1; j>=0; j--) {
        cout << "Bit pattern in byte " << j << ": ";
        for(i = 128; i; i >>= 1)
            if(i & c[j]) cout << "1";
            else cout << "0";
        cout << "\n";
    }
}

```



```
int main()
{
    bits ob(1991.829);

    ob.show_bits();

    return 0;
}
```

The output of this program is

```
Bit pattern in byte 7: 01000000
Bit pattern in byte 6: 10011111
Bit pattern in byte 5: 00011111
Bit pattern in byte 4: 01010000
Bit pattern in byte 3: 11100101
Bit pattern in byte 2: 01100000
Bit pattern in byte 1: 01000001
Bit pattern in byte 0: 10001001
```

3. Both structures and unions can have constructors and destructors. The following example shows the **strtype** class reworked as a structure. It contains both a constructor and a destructor function.

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

struct strtype {
    strtype(char *ptr);
    ~strtype();
    void show();
private:
    char *p;
    int len;
};

strtype::strtype(char *ptr)
{
    len = strlen(ptr);
    p = (char *) malloc(len+1);
```

C++

```

if(!p) {
    cout << "Allocation error\n";
    exit(1);
}
strcpy(p, ptr);
}

```

```

strtype::~strtype()
{
    cout << "Freeing p\n";
    free(p);
}

```

```

void strtype::show()
{
    cout << p << " - length: " << len;
    cout << "\n";
}

```

```

int main()
{
    strtype s1("This is a test."), s2("I like C++.");
    s1.show();
    s2.show();

    return 0;
}

```

4. This program uses an anonymous union to display the individual bytes that comprise a **double**. (This program assumes that **doubles** are 8 bytes long.)

```

// Using an anonymous union.
#include <iostream>
using namespace std;

```

```

int main()
{
    union {
        unsigned char bytes[8];
        double value;
    };
    int i;

```

```
    value = 859345.324;

    // display the bytes within a double
    for(i=0; i<8; i++)
        cout << (int) bytes[i] << " ";

    return 0;
}
```

As you can see, both **value** and **bytes** are accessed as if they were normal variables and not part of a union. Even though they are declared as being part of an anonymous union, their names are at the same scope level as any other local variable declared at the same point. This is why a member of an anonymous union cannot have the same name as any other variable known to its scope.

EXERCISES

1. Rewrite the **stack** class presented in Section 2.1 so it uses a structure rather than a class.
2. Use a **union** class to swap the low- and high-order bytes of an integer (assuming 16-bit integers; if your computer uses 32-bit integers, swap the bytes of a **short int**).
3. Explain what an anonymous union is and how it differs from a normal union.

2.6

IN-LINE FUNCTIONS

Before we continue this examination of classes, a short but related digression is needed. In C++, it is possible to define functions that are not actually called but, rather, are expanded in line, at the point of each call. This is much the same way that a C-like parameterized

C++

macro works. The advantage of *in-line functions* is that they have no overhead associated with the function call and return mechanism. This means that in-line functions can be executed much faster than normal functions. (Remember, the machine instructions that generate the function call and return take time each time a function is called. If there are parameters, even more time overhead is generated.)

The disadvantage of in-line functions is that if they are too large and called too often, your program grows larger. For this reason, in general only short functions are declared as in-line functions.

To declare an in-line function, simply precede the function's definition with the **inline** specifier. For example, this short program shows how to declare an in-line function:

```
// Example of an in-line function
#include <iostream>
using namespace std;

inline int even(int x)
{
    return !(x%2);
}

int main()
{
    if(even(10)) cout << "10 is even\n";
    if(even(11)) cout << "11 is even\n";
    return 0;
}
```

In this example, the function **even()**, which returns **true** if its argument is even, is declared as being in-line. This means that the line

```
if(even(10)) cout << "10 is even\n";
```

is functionally equivalent to

```
if(!(10%2)) cout << "10 is even\n";
```

This example also points out another important feature of using **inline**: an in-line function must be defined *before* it is first called. If it isn't, the compiler has no way to know that it is supposed to be expanded in-line. This is why **even()** was defined before **main()**.

The advantage of using **inline** rather than parameterized macros is twofold. First, it provides a more structured way to expand short functions in line. For example, when you are creating a parameterized macro, it is easy to forget that extra parentheses are often needed to ensure proper in-line expansion in every case. Using in-line functions prevents such problems.

Second, an in-line function might be able to be optimized more thoroughly by the compiler than a macro expansion. In any event, C++ programmers virtually never use parameterized macros, instead relying on **inline** to avoid the overhead of a function call associated with a short function.

It is important to understand that the **inline** specifier is a *request*, not a command, to the compiler. If, for various reasons, the compiler is unable to fulfill the request, the function is compiled as a normal function and the **inline** request is ignored.

Depending upon your compiler, several restrictions to in-line functions may apply. For example, some compilers will not in-line a function if it contains a **static** variable, a loop statement, a **switch** or a **goto**, or if the function is recursive. You should check your compiler's user manual for specific restrictions to in-line functions that might affect you.

If any in-line restriction is violated, the compiler is free to generate a normal function.

EXAMPLES

1. Any type of function can be in-lined, including functions that are members of classes. For example, here the member function **divisible()** is in-lined for fast execution. (The function returns **true** if its first argument can be evenly divided by its second.)

```
// Demonstrate in-lining a member function.
```

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

```
class samp {
    int i, j;
```

C++

```

public:
    samp(int a, int b);
    int divisible(); // in-lined in its definition
};

samp::samp(int a, int b)
{
    i = a;
    j = b;
}

/* Return 1 if i is evenly divisible by j.
   This member function is expanded in line.
*/
inline int samp::divisible()
{
    return !(i%j);
}

int main()
{
    samp ob1(10, 2), ob2(10, 3);

    // this is true
    if(ob1.divisible()) cout << "10 divisible by 2\n";

    // this is false
    if(ob2.divisible()) cout << "10 divisible by 3\n";

    return 0;
}

```

2. It is perfectly permissible to in-line an overloaded function. For example, this program overloads **min()** three ways. Each way is also declared as **inline**.

```

#include <iostream>
using namespace std;

// Overload min() three ways.

// integers
inline int min(int a, int b)
{
    return a<b ? a : b;
}

```

```

// longs
inline long min(long a, long b)
{
    return a < b ? a : b;
}

// doubles
inline double min(double a, double b)
{
    return a < b ? a : b;
}

int main()
{
    cout << min(-10, 10) << "\n";
    cout << min(-10.01, 100.002) << "\n";
    cout << min(-10L, 12L) << "\n";

    return 0;
}

```

EXERCISES

1. In Chapter 1 you overloaded the **abs()** function so that it could find the absolute value of integers, long integers, and **doubles**. Modify that program so that those functions are expanded in line.
2. Why might the following function not be in-lined by your compiler?

```

void f1()
{
    int i;

    for(i=0; i<10; i++) cout << i;
}

```

If a member function's definition is short enough, the definition can be included inside the class declaration. Doing so causes the function to automatically become an in-line function, if possible. When a function is defined within a class declaration, the **inline** keyword is no longer necessary. (However, it is not an error to use it in this situation.) For example, the **divisible()** function from the preceding section can be automatically in-lined as shown here:

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

class samp {
    int i, j;
public:
    samp(int a, int b);

    /* divisible() is defined here and automatically
       in-lined. */
    int divisible() { return !(i%j); }
};

samp::samp(int a, int b)
{
    i = a;
    j = b;
}

int main()
{
    samp ob1(10, 2), ob2(10, 3);

    // this is true
    if(ob1.divisible()) cout << "10 divisible by 2\n";

    // this is false
    if(ob2.divisible()) cout << "10 divisible by 3\n";

    return 0;
}
```

As you can see, the code associated with **divisible()** occurs inside the declaration for the class **samp**. Further notice that no other definition

of **divisible()** is needed—or permitted. Defining **divisible()** inside **samp** causes it to be made into an in-line function automatically.

When a function defined inside a class declaration cannot be made into an in-line function (because a restriction has been violated), it is automatically made into a regular function.

Notice how **divisible()** is defined within **samp**, paying particular attention to the body. It occurs all on one line. This format is very common in C++ programs when a function is declared within a class declaration. It allows the declaration to be more compact. However, the **samp** class could have been written like this:

```
class samp {
    int i, j;
public:
    samp(int a, int b);

    /* divisible() is defined here and automatically
       in-lined. */
    int divisible()
    {
        return !(i%j);
    }
};
```

In this version, the layout of **divisible()** uses the more-or-less standard indentation style. From the compiler's point of view, there is no difference between the compact style and the standard style. However, the compact style is commonly found in C++ programs when short functions are defined inside a class definition.

The same restrictions that apply to "normal" in-line functions apply to automatic in-line functions within a class declaration.

EXAMPLES

1. Perhaps the most common use of in-line functions defined within a class is to define constructor and destructor functions. For example, the **samp** class can more efficiently be defined like this:

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

C++

```

class samp {
    int i, j;
public:
    // inline constructor
    samp(int a, int b) { i = a; j = b; }
    int divisible() { return !(i%j); }
};

```

The definition of **samp()** within the class **samp** is sufficient, and no other definition of **samp()** is needed.

2. Sometimes a short function will be included in a class declaration even though the automatic in-lining feature is of little or no value. Consider this class declaration:

```

class myclass {
    int i;
public:
    myclass(int n) { i = n; }
    void show() { cout << i; }
};

```

Here the function **show()** is made into an in-line function automatically. However, as you should know, I/O operations are (generally) so slow relative to CPU/memory operations that any effect of eliminating the function call overhead is essentially lost. Even so, in C++ programs, it is still common to see small functions of this type declared within a class simply for the sake of convenience, and because no harm is caused.

EXERCISES

1. Convert the **stack** class from Section 2.1, Example 1, so that it uses automatic in-line functions where appropriate.
2. Convert the **strtype** class from Section 2.2, Example 3, so that it uses automatic in-line functions.

SKILLS CHECK



Mastery
Skills Check

At this point you should be able to perform the following exercises and answer the questions.

1. What is a constructor? What is a destructor? When are they executed?
2. Create a class called **line** that draws a line on the screen. Store the line length in a private integer variable called **len**. Have **line**'s constructor take one parameter: the line length. Have the constructor store the length and actually draw the line. If your system does not support graphics, display the line by using *****. Optional: Give **line** a destructor that erases the line.
3. What does the following program display?

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

int main()
{
    int i = 10;
    long l = 1000000;
    double d = -0.0009;

    cout << i << ' ' << l << ' ' << d;
    cout << "\n";

    return 0;
}
```

4. Add another derived class that inherits **area_cl** from Section 2.3, Exercise 1. Call this class **cylinder** and have it compute the surface area of a cylinder. Hint: The surface area of a cylinder is $2 * pi * R^2 + pi * D * height$.

5. What is an in-line function? What are its advantages and disadvantages?
6. Modify the following program so that all member functions are automatically in-lined:

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

class myclass {
    int i, j;
public:
    myclass(int x, int y);
    void show();
};

myclass::myclass(int x, int y)
{
    i = x;
    j = y;
}

void myclass::show()
{
    cout << i << " " << j << "\n";
}

int main()
{
    myclass count(2, 3);

    count.show();

    return 0;
}
```

7. What is the difference between a class and a structure?
8. Is the following fragment valid?

```
union {
    float f;
    unsigned int bits;
};
```



This section checks how well you have integrated material in this chapter with that from the preceding chapter.

1. Create a class called **prompt**. Pass its constructor function a prompting string of your own choosing. Have the constructor display the string and then input an integer. Store this value in a private variable called **count**. When an object of type **prompt** is destroyed, ring the bell on the terminal as many times as the user entered.
2. In Chapter 1 you created a program that converted feet to inches. Now create a class that does the same thing. Have the class store the number of feet and its equivalent number of inches. Pass to the class's constructor the number of feet and have the constructor display the number of inches.
3. Create a class called **dice** that contains one private integer variable. Create a function called **roll()** that uses the standard random number generator, **rand()**, to generate a number between 1 and 6. Then have **roll()** display that value.



The student can do how well you have provided material in this chapter will be the preceding chapter.

The student can do how well you have provided material in this

chapter will be the preceding chapter.

The student can do how well you have provided material in this

chapter will be the preceding chapter.

The student can do how well you have provided material in this

chapter will be the preceding chapter.

The student can do how well you have provided material in this

chapter will be the preceding chapter.

The student can do how well you have provided material in this

chapter will be the preceding chapter.

The student can do how well you have provided material in this

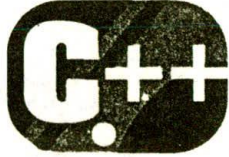
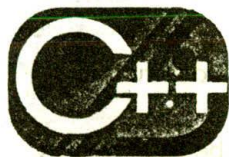
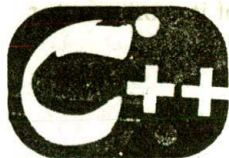
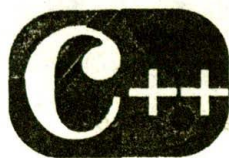
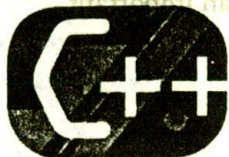
chapter will be the preceding chapter.



3

*A Closer Look
at Classes***chapter objectives**

- 3.1** Assigning objects
- 3.2** Passing objects to functions
- 3.3** Returning objects from functions
- 3.4** An Introduction to friend functions



In this chapter you continue to explore the class. You will learn about assigning objects, passing objects to functions, and returning objects from functions. You will also learn about an important new type of function: the friend.



Before proceeding, you should be able to correctly answer the following questions and do the exercises.

1. Given the following class, what are the names of its constructor and destructor functions?

```
class widgit {
    int x, y;
public:
    // ... fill in constructor and destructor functions
};
```

2. When is a constructor function called? When is a destructor function called?
3. Given the following base class, show how it can be inherited by a derived class called **Mars**.

```
class planet {
    int moons;
    double dist_from_sun;
    double diameter;
    double mass;
public:
    // ...
};
```

4. There are two ways to cause a function to be expanded in line. What are they?
5. Give two possible restrictions to in-line functions.
6. Given the following class, show how an object called **ob** that passes the value 100 to **a** and X to **c** would be declared.


```
class sample {
    int a;
    char c;
public:
    sample(int x, char ch) { a = x; c = ch; }
    // ...
};
```

3.1

ASSIGNING OBJECTS

One object can be assigned to another provided that both objects are of the same type. By default, when one object is assigned to another, a bitwise copy of all the data members is made. For example, when an object called **o1** is assigned to another object called **o2**, the contents of all of **o1**'s data are copied into the equivalent members of **o2**. This is illustrated by the following program:

```
// An example of object assignment.
#include <iostream>
using namespace std;

class myclass {
    int a, b;
public:
    void set(int i, int j) { a = i; b = j; }
    void show() { cout << a << ' ' << b << "\n"; }
};

int main()
{
    myclass o1, o2;

    o1.set(10, 4);

    // assign o1 to o2
    o2 = o1;

    o1.show();
    o2.show();

    return 0;
}
```

Here, object **o1** has its member variables **a** and **b** set to the values 10 and 4, respectively. Next, **o1** is assigned to **o2**. This causes the current value of **o1.a** to be assigned to **o2.a** and **o1.b** to be assigned to **o2.b**. Thus, when run, this program displays

```
10 4
10 4
```

Keep in mind that an assignment between two objects simply makes the data in those objects identical. The two objects are still completely separate. For example, after the assignment, calling **o1.set()** to set the value of **o1.a** has no effect on **o2** or its **a** value.

EXAMPLES

1. Only objects of the same type can be used in an assignment statement. If the objects are not of the same type, a compile-time error is reported. Further, it is not sufficient that the types just be physically similar—their type names must be the same. For example, this is not a valid program:

```
// This program has an error.
#include <iostream>
using namespace std;

class myclass {
    int a, b;
public:
    void set(int i, int j) { a = i; b = j; }
    void show() { cout << a << ' ' << b << "\n"; }
};

/* This class is similar to myclass but uses a
   different class name and thus appears as a different
   type to the compiler.
*/
class yourclass {
    int a, b;
public:
    void set(int i, int j) { a = i; b = j; }
    void show() { cout << a << ' ' << b << "\n"; }
};
```

```
int main()
{
    myclass o1;
    yourclass o2;

    o1.set(10, 4);

    o2 = o1; // ERROR, objects not of same type

    o1.show();
    o2.show();

    return 0;
}
```

In this case, even though **myclass** and **yourclass** are physically the same, because they have different type names, they are treated as differing types by the compiler.

2. It is important to understand that all data members of one object are assigned to another when an assignment is performed. This includes compound data such as arrays. For example, in the following version of the **stack** example, only **s1** has any characters actually pushed onto it. However, because of the assignment, **s2's stack** array will also contain the characters **a**, **b**, and **c**.

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

#define SIZE 10

// Declare a stack class for characters.
class stack {
    char stck[SIZE]; // holds the stack
    int tos; // index of top of stack
public:
    stack(); // constructor
    void push(char ch); // push character on stack
    char pop(); // pop character from stack
};

// Initialize the stack.
stack::stack()
{
```

C++

```
cout << "Constructing a stack\n";
tos = 0;
}

// Push a character.
void stack::push(char ch)
{
    if(tos==SIZE) {
        cout << "Stack is full\n";
        return;
    }
    stck[tos] = ch;
    tos++;
}

// Pop a character.
char stack::pop()
{
    if(tos==0) {
        cout << "Stack is empty\n";
        return 0; // return null on empty stack
    }
    tos--;
    return stck[tos];
}

int main()
{
    // Create two stacks that are automatically initialized.
    stack s1, s2;
    int i;

    s1.push('a');
    s1.push('b');
    s1.push('c');

    // clone s1
    s2 = s1; // now s1 and s2 are identical

    for(i=0; i<3; i++) cout << "Pop s1: " << s1.pop() << "\n";
```

```
for(i=0; i<3; i++) cout << "Pop s2: " << s2.pop() << "\n";
```

```
return 0;
```

```
}
```

3. You must exercise some care when assigning one object to another. For example, here is the **strtype** class developed in Chapter 2, along with a short **main()**. See if you can find an error in this program.

```
// This program contains an error.
```

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

```
class strtype {
    char *p;
    int len;
public:
    strtype(char *ptr);
    ~strtype();
    void show();
};
```

```
strtype::strtype(char *ptr)
```

```
{
    len = strlen(ptr);
    p = (char *) malloc(len+1);
    if(!p) {
        cout << "Allocation error\n";
        exit(1);
    }
```

```
    strcpy(p, ptr);
}
```

```
strtype::~strtype()
```

```
{
    cout << "Freeing p\n";
    free(p);
}
```

```
    }

void strtype::show()
{
    cout << p << " - length: " << len;
    cout << "\n";
}

int main()
{
    strtype s1("This is a test."), s2("I like C++.");

    s1.show();
    s2.show();

    // assign s1 to s2 - - this generates an error
    s2 = s1;

    s1.show();
    s2.show();

    return 0;
}
```

The trouble with this program is quite insidious. When **s1** and **s2** are created, both allocate memory to hold their respective strings. A pointer to each object's allocated memory is stored in **p**. When a **strtype** object is destroyed, this memory is released. However, when **s1** is assigned to **s2**, **s2's p** now points to the same memory as **s1's p**. Thus, when these objects are destroyed, the memory pointed to by **s1's p** is freed *twice* and the memory originally pointed to by **s2's p** is not freed *at all*.

While benign in this context, this sort of problem occurring in a real program will cause the dynamic allocation system to fail, and possibly even cause a program crash. As you can see from the preceding example, when assigning one object to another, you must make sure you are not destroying information that may be needed later.

EXERCISES

1. What is wrong with the following fragment?

```
// This program has an error.
#include <iostream>
using namespace std;

class c11 {
    int i, j;
public:
    c11(int a, int b) { i = a; j = b; }
    // ...
};

class c12 {
    int i, j;
public:
    c12(int a, int b) { i = a; j = b; }
    // ...
};

int main()
{
    c11 x(10, 20);
    c12 y(0, 0);
    x = y;

    // ...
}
```

2. Using the **queue** class that you created for Chapter 2, Section 2.1, Exercise 1, show how one **queue** can be assigned to another.
3. If the **queue** class from the preceding question dynamically allocates memory to hold the queue, why, in this situation, can one **queue** not be assigned to another?
-

3.2

PASSING OBJECTS TO FUNCTION

Objects can be passed to functions as arguments in just the same way that other types of data are passed. Simply declare the function's parameter as a class type and then use an object of that class as an argument when calling the function. As with other types of data, by default all objects are passed by value to a function.

EXAMPLES

- Here is a short example that passes an object to a function:

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

class samp {
    int i;
public:
    samp(int n) { i = n; }
    int get_i() { return i; }
};

// Return square of o.i.
int sqr_it(samp o)
{
    return o.get_i() * o.get_i();
}

int main()
{
    samp a(10), b(2);
    cout << sqr_it(a) << "\n";
    cout << sqr_it(b) << "\n";
    return 0;
}
```

This program creates a class called **samp** that contains one integer variable called **i**. The function **sqr_it()** takes an argument of type **samp** and returns the square of that object's value. The output from this program is 100 followed by 4.

As stated, the default method of parameter passing in C++, including objects, is by value. This means that a bitwise copy of the argument is made and it is this copy that is used by the function. Therefore, changes to the object inside the function do not affect the calling object. This is illustrated by the following example:

```
/*
   Remember, objects, like other parameters, are passed
   by value. Thus changes to the parameter inside a
   function have no effect on the object used in the call.
*/
#include <iostream>
using namespace std;

class samp {
    int i;
public:
    samp(int n) { i = n; }
    void set_i(int n) { i = n; }
    int get_i() { return i; }
};

/* Set o.i to its square. This has no effect on the
   object used to call sqr_it(), however.
*/
void sqr_it(samp o)
{
    o.set_i(o.get_i() * o.get_i());

    cout << "Copy of a has i value of " << o.get_i();
    cout << "\n";
}

int main()
{
    samp a(10);

    sqr_it(a); // a passed by value

    cout << "But, a.i is unchanged in main: ";
    cout << a.get_i(); // displays 10

    return 0;
}
```

The output displayed by this program is

```
Copy of a has i value of 100
But, a.i is unchanged in main: 10
```

3. As with other types of variables, the address of an object can be passed to a function so that the argument used in the call can be modified by the function. For example, the following version of the program in the preceding example does, indeed, modify the value of the object whose address is used in the call to `sqr_it()`.

```
/*
   Now that the address of an object is passed to sqr_it(),
   the function can modify the value of the argument whose
   address is used in the call.
*/
#include <iostream>
using namespace std;

class samp {
    int i;
public:
    samp(int n) { i = n; }
    void set_i(int n) { i = n; }
    int get_i() { return i; }
};

/* Set o.i to its square. This affects the calling
   argument.
*/
void sqr_it(samp *o)
{
    o->set_i(o->get_i() * o->get_i());

    cout << "Copy of a has i value of " << o->get_i();
    cout << "\n";
}

int main()
{
    samp a(10);

    sqr_it(&a); // pass a's address to sqr_it()

    cout << "Now, a in main() has been changed: ";
    cout << a.get_i(); // displays 100
}
```

```
return 0;
```

This program now displays the following output:

```
Copy of a has i value of 100
Now, a in main() has been changed: 100
```

4. When a copy of an object is made when being passed to a function, it means that a new object comes into existence. Also, when the function that the object was passed to terminates, the copy of the argument is destroyed. This raises two 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 answer may, at first, seem surprising.

When a copy of an object is made to be used in a function call, the constructor function is *not* called. The reason for this is simple to understand if you think about it. Since a constructor function is generally used to initialize some aspect of an object, it must not be called when making a copy of an already existing object passed to a function. Doing so would alter the contents of the object. When passing an object to a function, you want the current state of the object, not its initial state.

However, when the function terminates and the copy is destroyed, the destructor function *is* called. This is because the object might perform some operation that must be undone when it goes out of scope. For example, the copy may allocate memory that must be released.

To summarize, when a copy of an object is created because it is used as an argument to a function, the constructor function is not called. However, when the copy is destroyed (usually by going out of scope when the function returns), the destructor function is called.

The following program illustrates the preceding discussion:

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

```
class samp {
    int i;
public:
    samp(int n) {
        i = n;
```

C++

```

    cout << "Constructing\n";
}
~samp() { cout << "Destructing\n"; }
int get_i() { return i; }
};

// Return square of o.i.
int sqr_it(samp o)
{
    return o.get_i() * o.get_i();
}

int main()
{
    samp a(10);

    cout << sqr_it(a) << "\n";

    return 0;
}

```

This function displays the following:

```

Constructing
Destructing
100
Destructing

```

As you can see, only one call to the constructor function is made. This occurs when **a** is created. However, two calls to the destructor are made. One is for the copy created when **a** is passed to **sqr_it()**. The other is for **a** itself.

The fact that the destructor for the object that is the copy of the argument is executed when the function terminates can be a source of problems. For example, if the object used as the argument allocates dynamic memory and frees that memory when destroyed, its copy will free the same memory when its destructor is called. This will leave the original object damaged and effectively useless. (See Exercise 2, just ahead in this section, for an example.) It is important to guard against this type of error and to make sure that the destructor function of the copy of an object used in an argument does not cause side effects that alter the original argument.

As you might guess, one way around the problem of a parameter's destructor function destroying data needed by the calling argument is to pass the address of the object and not the object itself. When an address is passed, no new object is created, and therefore, no destructor is called when the function returns. (As you will see in the next chapter, C++ provides a variation on this theme that offers a very elegant alternative.) However, an even better solution exists, which you can use after you have learned about a special type of constructor called a *copy constructor*. A copy constructor lets you define precisely how copies of objects are made. (Copy constructors are discussed in Chapter 5.)

EXERCISES

1. Using the **stack** example from Section 3.1, Example 2, add a function called **showstack()** that is passed an object of type **stack**. Have this function display the contents of a stack.
2. As you know, when an object is passed to a function, a copy of that object is made. Further, when that function returns, the copy's destructor function is called. Keeping this in mind, what is wrong with the following program?

```
// This program contains an error.
```

```
#include <iostream>
```

```
#include <cstdlib>
```

```
using namespace std;
```

```
class dyna {
```

```
    int *p;
```

```
public:
```

```
    dyna(int i);
```

```
    ~dyna() { free(p); cout << "freeing \n"; }
```

```
    int get() { return *p; }
```

```
};
```

```
dyna::dyna(int i)
```

```
{
```

```
    p = (int *) malloc(sizeof(int));
```

```
    if(!p) {
```

```

    cout << "Allocation failure\n";
    exit(1);
}

*p = i;
}

// Return negative value of *ob.p
int neg(dyna ob)
{
    return -ob.get();
}

int main()
{
    dyna o(-10);

    cout << o.get() << "\n";
    cout << neg(o) << "\n";

    dyna o2(20);
    cout << o2.get() << "\n";
    cout << neg(o2) << "\n";

    cout << o.get() << "\n";
    cout << neg(o) << "\n";

    return 0;
}

```

3.3

RETURNING OBJECTS FROM FUNCTIONS

Just as you can pass objects to functions, functions can return objects. To do so, first declare the function as returning a class type. Second, return an object of that type using the normal **return** statement.

There is one important point to understand about returning objects from functions, however: When an object is returned by a function, a temporary object is automatically created which 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 might cause unexpected side effects in some situations, as is illustrated in Example 2 below.

EXAMPLES

1. Here is an example of a function that returns an object:

```
// Returning an object
#include <iostream>
#include <cstring>
using namespace std;

class samp {
    char s[80];
public:
    void show() { cout << s << "\n"; }
    void set(char *str) { strcpy(s, str); }
};

// Return an object of type samp
samp input()
{
    char s[80];
    samp str;

    cout << "Enter a string: ";
    cin >> s;

    str.set(s);

    return str;
}

int main()
{
    samp ob;

    // assign returned object to ob
    ob = input();
    ob.show();

    return 0;
}
```

C++

In this example, `input()` creates a local object called `str` and then reads a string from the keyboard. This string is copied into `str.s`, and then `str` is returned by the function. This object is then assigned to `ob` inside `main()` when it is returned by the call to `input()`.

2. You must be careful about returning objects from functions if those objects contain destructor functions because the returned object goes out of scope as soon as the value is returned to the calling routine. 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 assigned the return value is still using it. For example, consider this incorrect version of the preceding program:

```
// An error generated by returning an object.
#include <iostream>
#include <cstring>
#include <cstdlib>
using namespace std;

class samp {
    char *s;
public:
    samp() { s = '\0'; }
    ~samp() { if(s) free(s); cout << "Freeing s\n"; }
    void show() { cout << s << "\n"; }
    void set(char *str);
};

// Load a string.
void samp::set(char *str)
{
    s = (char *) malloc(strlen(str)+1);
    if(!s) {
        cout << "Allocation error\n";
        exit(1);
    }

    strcpy(s, str);
}

// Return an object of type samp.
samp input()
```



```

char s[80];
samp str;

cout << "Enter a string: ";
cin >> s;

str.set(s);
return str;
}

int main()
{
    samp ob;

    // assign returned object to ob
    ob = input(); // This causes an error!!!!
    ob.show();

    return 0;
}

```

The output from this program is shown here:

```

Enter a string: Hello
Freeing s
Freeing s
Hello
Freeing s
Null pointer assignment

```

Notice that **samp**'s destructor function is called three times. First, it is called when the local object **str** goes out of scope when **input()** returns. The second time **~samp()** is called is when the temporary object returned by **input()** is destroyed. Remember, when an object is returned from a function, an invisible (to you) temporary object is automatically generated which holds the return value. In this case, this object is simply a copy of **str**, which is the return value of the function. Therefore, after the function has returned, the temporary object's destructor is executed. Finally, the destructor for object **ob**, inside **main()**, is called when the program terminates.

The trouble is that in this situation, the first time the destructor executes, the memory allocated to hold the string

input by `input()` is freed. Thus, not only do the other two calls to `samp`'s destructor try to free an already released piece of dynamic memory, but they destroy the dynamic allocation system in the process, as evidenced by the run-time message "Null pointer assignment." (Depending upon your compiler, the memory model used for compilation, and the like, you may or may not see this message if you try this program.)

The key point to be understood from this example is that when an object is returned from a function, the temporary object used to effect the return will have its destructor function called. Thus, you should avoid returning objects in which this situation is harmful. (As you will learn in Chapter 5, it is possible to use a copy constructor to manage this situation.)

EXERCISES

1. To illustrate exactly when an object is constructed and destructed when returned from a function, create a class called `who`. Have `who`'s constructor take one character argument that will be used to identify an object. Have the constructor display a message similar to this when constructing an object:

```
Constructing who #x
```

where `x` is the identifying character associated with each object. When an object is destroyed, have a message similar to this displayed:

```
Destroying who #x
```

where, again, `x` is the identifying character. Finally, create a function called `make_who()` that returns a `who` object. Give each object a unique name. Note the output displayed by the program.

2. Other than the incorrect freeing of dynamically allocated memory, think of a situation in which it would be improper to return an object from a function.
-

AN INTRODUCTION TO FRIEND FUNCTIONS

There will be times when you want a function to have access to the private members of a class without that function actually being a member of that class. Towards this end, C++ supports friend functions (A friend is not a member of a class but still has access to its private elements.)

(Two reasons that friend functions are useful have to do with operator overloading and the creation of certain types of I/O functions.) You will have to wait until later to see these uses of a friend in action. However, a third reason for friend functions is that there will be times when you want one function to have access to the private members of *two or more* different classes. It is this use that is examined here.

A friend function is defined as a regular, nonmember function. However, inside the class declaration for which it will be a friend, its prototype is also included, prefaced by the keyword **friend**. To understand how this works, examine this short program:

```
// An example of a friend function.
#include <iostream>
using namespace std;

class myclass {
    int n, d;
public:
    myclass(int i, int j) { n = i; d = j; }
    // declare a friend of myclass
    friend int isfactor(myclass ob);
};

/* Here is friend function definition. It returns true
   if d is a factor of n. Notice that the keyword
   friend is not used in the definition of isfactor().
*/
int isfactor(myclass ob)
{
    if(!(ob.n % ob.d)) return 1;
    else return 0;
}
```

```
int main()
{
    myclass ob1(10, 2), ob2(13, 3);

    if(isfactor(ob1)) cout << "2 is a factor of 10\n";
    else cout << "2 is not a factor of 10\n";

    if(isfactor(ob2)) cout << "3 is a factor of 13\n";
    else cout << "3 is not a factor of 13\n";

    return 0;
}
```

In this example, **myclass** declares its constructor function and the friend **isfactor()** inside its class declaration. Because **isfactor()** is a friend of **myclass**, **isfactor()** has access to its private members. This is why, within **isfactor()**, it is possible to directly refer to **ob.n** and **ob.d**.

It is important to understand that a friend function is not a member of the class for which it is a friend. Thus, it is not possible to call a friend function by using an object name and a class member access operator (a dot or an arrow). For example, given the preceding example, this statement is wrong:

```
ob1.isfactor(); // wrong; isfactor() is not a member function
```

Instead, friends are called just like regular functions.

Although a friend function has knowledge of the private elements of the class for which it is a friend, it can only access them through an object of the class. That is, unlike a member function of **myclass**, which can refer to **n** or **d** directly, a friend can access these variables only in conjunction with an object that is declared within or passed to the friend function.

The preceding paragraph brings up an important side issue. When a member function refers to a private element, it does so directly because a member function is executed only in conjunction with an object of that class. Thus, when a member function refers to a private element, the compiler knows which object that private element belongs to by the object that is linked to the function when that member function is called. However, a friend function is not linked to any object. It simply is granted access to the private elements of a class. Thus, inside the friend function, it is meaningless to refer to a private member without reference to a specific object.

Because friends are not members of a class, they will typically be passed one or more objects of the class for which they are friends. This is the case with `isfactor()`. It is passed an object of `myclass`, called `ob`. However, because `isfactor()` is a friend of `myclass`, it can access `ob`'s private elements. If `isfactor()` had not been made a friend of `myclass`, it would not be able to access `ob.d` or `ob.n` since `n` and `d` are private members of `myclass`.

A friend function is not a member and cannot be qualified by an object name. It must be called just like a normal function.

A friend function is not inherited. That is, when a base class includes a friend function, that friend function is not a friend of a derived class.

One other important point about friend functions is that a friend function can be friends with more than one class.

EXAMPLES

1. One common (and good) use of a friend function occurs when two different types of classes have some quantity in common that needs to be compared. For example, consider the following program, which creates a class called `car` and a class called `truck`, each containing, as a private variable, the speed of the vehicle it represents:

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

```
class truck; // a forward declaration
```

```
class car {
    int passengers;
    int speed;
public:
    car(int p, int s) { passengers = p; speed = s; }
    friend int sp_greater(car c, truck t);
};
```

```
class truck {
    int weight;
```

```
int speed;
```

```
public:
```

```
truck(int w, int s) { weight = w; speed = s; }
```

```
friend int sp_greater(car c, truck t);
```

```
};
```

```
/* Return positive if car speed faster than truck.
```

```
Return 0 if speeds are the same.
```

```
Return negative if truck speed faster than car.
```

```
*/
```

```
int sp_greater(car c, truck t)
```

```
{
```

```
return c.speed - t.speed;
```

```
}
```

```
int main()
```

```
{
```

```
int t;
```

```
car c1(6, 55), c2(2, 120);
```

```
truck t1(10000, 55), t2(20000, 72);
```

```
cout << "Comparing c1 and t1:\n";
```

```
t = sp_greater(c1, t1);
```

```
if(t < 0) cout << "Truck is faster.\n";
```

```
else if(t == 0) cout << "Car and truck speed is the same.\n";
```

```
else cout << "Car is faster.\n";
```

```
cout << "\nComparing c2 and t2:\n";
```

```
t = sp_greater(c2, t2);
```

```
if(t < 0) cout << "Truck is faster.\n";
```

```
else if(t == 0) cout << "Car and truck speed is the same.\n";
```

```
else cout << "Car is faster.\n";
```

```
return 0;
```

```
}
```

This program contains the function `sp_greater()`, which is a friend function of both the `car` and `truck` classes. (As stated, a function can be a friend of two or more classes.) This function returns positive if the `car` object is going faster than the `truck` object, 0 if their speeds are the same, and negative if the `truck` is going faster.

This program illustrates one important C++ syntax element: the *forward declaration* (also called a *forward reference*). Because

`sp_greater()` takes parameters of both the `car` and the `truck` classes, it is logically impossible to declare both before including `sp_greater()` in either. Therefore, there needs to be some way to tell the compiler about a class name without actually declaring it. This is called a forward declaration. In C++, to tell the compiler that an identifier is the name of a class, use a line like this before the class name is first used:

```
class class-name;
```

For example, in the preceding program, the forward declaration is

```
class truck;
```

Now `truck` can be used in the friend declaration of `sp_greater()` without generating a compile-time error.

2. A function can be a member of one class and a friend of another. For example, here is the preceding example rewritten so that `sp_greater()` is a member of `car` and a friend of `truck`:

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

class truck; // a forward declaration

class car {
    int passengers;
    int speed;
public:
    car(int p, int s) { passengers = p; speed = s; }
    int sp_greater(truck t);
};

class truck {
    int weight;
    int speed;
public:
    truck(int w, int s) { weight = w; speed = s; }

    // note new use of the scope resolution operator
    friend int car::sp_greater(truck t);
};
```

C++

```

/* Return positive if car speed faster than truck.
   Return 0 if speeds are the same.
   Return negative if truck speed faster than car.
*/
int car::sp_greater(truck t)
{
    /* Since sp_greater() is member of car, only a
       truck object must be passed to it. */

    return speed-t.speed;
}

int main()
{
    int t;
    car c1(6, 55), c2(2, 120);
    truck t1(10000, 55), t2(20000, 72);

    cout << "Comparing c1 and t1:\n";
    t = c1.sp_greater(t1); // evoke as member function of car
    if(t<0) cout << "Truck is faster.\n";
    else if(t==0) cout << "Car and truck speed is the same.\n";
    else cout << "Car is faster.\n";

    cout << "\nComparing c2 and t2:\n";
    t = c2.sp_greater(t2); // evoke as member function of car
    if(t<0) cout << "Truck is faster.\n";
    else if(t==0) cout << "Car and truck speed is the same.\n";
    else cout << "Car is faster.\n";

    return 0;
}

```

Notice the new use of the scope resolution operator as it occurs in the friend declaration within the **truck** class declaration. In this case, it is used to tell the compiler that the function **sp_greater()** is a member of the **car** class.

One easy way to remember how to use the scope resolution operator is that the class name followed by the scope resolution operator followed by the member name fully specifies a class member.

In fact, when referring to a member of a class, it is never wrong to fully specify its name. However, when an object is

used to call a member function or access a member variable, the full name is redundant and seldom used. For example,

```
t = c1.sp_greater(t1);
```

can be written using the (redundant) scope resolution operator and the class name **car** like this:

```
t = c1.car::sp_greater(t1);
```

However, since **c1** is an object of type **car**, the compiler already knows that **sp_greater()** is a member of the **car** class, making the full class specification unnecessary.

EXERCISE

1. Imagine a situation in which two classes, called **pr1** and **pr2**, shown here, share one printer. Further, imagine that other parts of your program need to know when the printer is in use by an object of either of these two classes. Create a function called **inuse()** that returns **true** when the printer is being used by either and **false** otherwise. Make this function a friend of both **pr1** and **pr2**.

```
class pr1 {
    int printing;
    // ...
public:
    pr1() { printing = 0; }
    void set_print(int status) { printing = status; }
    // ...
};
```

```
class pr2 {
    int printing;
    // ...
public:
    pr2() { printing = 0; }
    void set_print(int status) { printing = status; }
    // ...
};
```

SKILLS CHECK



Mastery

Skills Check

Before proceeding, you should be able to answer the following questions and perform the exercises.

1. What single prerequisite must be met in order for one object to be assigned to another?
2. Given this class fragment,

```
class samp {
    double *p;
public:
    samp(double d) {
        p = (double *) malloc(sizeof(double));
        if(!p) exit(1); // allocation error
        *p = d;
    }
    ~samp() { *free(p); }
    // ...
};

// ...
samp ob1(123.09), ob2(0.0);
// ...
ob2 = ob1;
```

what problem is caused by the assignment of **ob1** to **ob2**?

3. Given this class,

```
class planet {
    int moons;
    double dist_from_sun; // in miles
    double diameter;
    double mass;
public:
    //...
    double get_miles() { return dist_from_sun; }
};
```

create a function called **light()** that takes as an argument an object of type **plane** and returns the number of seconds that it takes light from the sun to reach the planet. (Assume that light travels at 186,000 miles per second and that **dist_from_sun** is specified in miles.)

- Can the address of an object be passed to a function as an argument?
- Using the **stack** class, write a function called **loadstack()** that returns a stack that is already loaded with the letters of the alphabet (a-z). Assign this stack to another object in the calling routine and prove that it contains the alphabet. Be sure to change the stack size so it is large enough to hold the alphabet.
- Explain why you must be careful when passing objects to a function or returning objects from a function.
- What is a friend function?



This section checks how well you have integrated the material in this chapter with that from earlier chapters.

- Functions can be overloaded as long as the number or type of their parameters differs. Overload **loadstack()** from Exercise 5 of the Mastery Skills Check so that it takes an integer, called **upper**, as a parameter. In the overloaded version, if **upper** is 1, load the stack with the uppercase alphabet. Otherwise, load it with the lowercase alphabet.
- Using the **strtype** class shown in Section 3.1, Example 3, add a friend function that takes as an argument a pointer to an object of type **strtype** and returns a pointer to the string pointed to by that object. (That is, have the function return **p**.) Call this function **get_string()**.

3. Experiment: When an object of a derived class is assigned to another object of the same derived class, is the data associated with the base class also copied? To find out, use the following two classes and write a program that demonstrates what happens.

```
class base {
    int a;
public:
    void load_a(int n) { a = n; }
    int get_a() { return a; }
};

class derived : public base {
    int b;
public:
    void load_b(int n) { b = n; }
    int get_b() { return b; }
};
```

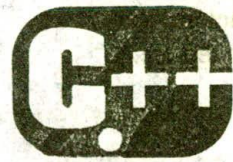
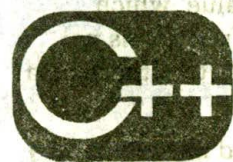
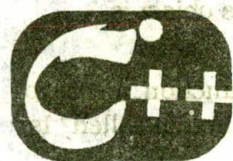
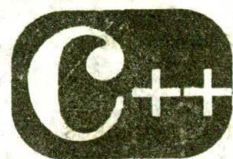
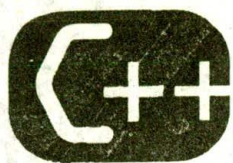


4

Arrays, Pointers, and References

chapter objectives

- 4.1 Arrays of objects
- 4.2 Using pointers to objects
- 4.3 The **this** pointer
- 4.4 Using **new** and **delete**
- 4.5 More about **new** and **delete**
- 4.6 References
- 4.7 Passing references to objects
- 4.8 Returning references
- 4.9 Independent references and restrictions



THIS chapter examines several important issues involving arrays of objects and pointers to objects. It concludes with a discussion of one of C++'s most important innovations: the reference. The reference is crucial to many C++ features, so a careful reading is advised.



Before proceeding, you should be able to correctly answer the following questions and do the exercises.

1. When one object is assigned to another, what precisely takes place?
2. Can any troubles or side effects occur when one object is assigned to another? (Give an example.)
3. When an object is passed as an argument to a function, a copy of that object is made. Is the copy's constructor function called? Is its destructor called?
4. By default, objects are passed to functions by value, which means that what occurs to the copy inside the function is not supposed to affect the argument used in the call. Can there be a violation of this principle? If so, give an example.
5. Given the following class, create a function called **make_sum()** that returns an object of type **summation**. Have this function prompt the user for a number and then construct an object having this value and return it to the calling procedure. Demonstrate that the function works.

```
class summation {
    int num;
    long sum; // summation of num
public:
    void set_sum(int n);
    void show_sum() {
        cout << num << " summed is " << sum << "\n";
    }
}
```

```
};

void summation::set_sum(int n)
{
    int i;
    num = n;

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

- In the preceding question, the function `set_sum()` was not defined in line within the `summation` class declaration. Give a reason why this might be necessary for some compilers.
- Given the following class, show how to add a friend function called `isneg()` that takes one parameter of type `myclass` and returns true if `num` is negative and false otherwise.

```
class myclass {
    int num;
public:
    myclass(int x) { num = x; }
};
```

- Can a friend function be friends with more than one class?

4.1

ARRAYS OF OBJECTS

As has been stated several times, objects are variables and have the same capabilities and attributes as any other type of variable. Therefore, it is perfectly acceptable for objects to be arrayed. The syntax for declaring an array of objects is exactly like that used to declare an array of any other type of variable. Further, arrays of objects are accessed just like arrays of other types of variables.

EXAMPLES

- Here is an example of an array of objects:

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

```

class samp {
    int a;
public:
    void set_a(int n) { a = n; }
    int get_a() { return a; }
};

int main()
{
    samp ob[4];
    int i;
    for(i=0; i<4; i++) ob[i].set_a(i);
    for(i=0; i<4; i++) cout << ob[i].get_a( );
    cout << "\n";
    return 0;
}

```

This program creates a four-element array of objects of type **samp** and then loads each element's **a** with a value between 0 and 3. Notice how member functions are called relative to each array element. The array name, in this case **ob**, is indexed; then the member access operator is applied, followed by the name of the member function to be called.

2. If a class type includes a constructor, an array of objects can be initialized. For example, here **ob** is an initialized array:

```

// Initialize an array
#include <iostream>
using namespace std;

class samp {
    int a;
public:
    samp(int n) { a = n; }
    int get_a() { return a; }
};

int main()
{
    samp ob[4] = { -1, -2, -3, -4 };
}

```



```

int i;

for(i=0; i<4; i++) cout << ob[i].get_a() << ' ';

cout << "\n";

return 0;
}

```

This program displays **-1 -2 -3 -4** on the screen. In this example, the values -1 through -4 are passed to the **ob** constructor function.

Actually, the syntax shown in the initialization list is shorthand for this longer form (first shown in Chapter 2):

```

samp ob[4] = { samp(-1), samp(- 2),
              samp(-3), samp(- 4) };

```

However, the form used in the program is more common (although, as you will see, this form will work only with arrays whose constructors take only one argument).

3. You can also have multidimensional arrays of objects. For example, here is a program that creates a two-dimensional array of objects and initializes them:

```

// Create a two-dimensional array of objects.
#include <iostream>
using namespace std;

class samp {
    int a;
public:
    samp(int n) { a = n; }
    int get_a() { return a; }
};

int main()
{
    samp ob[4][2] = {
        1, 2,
        3, 4,
        5, 6,
        7, 8
    };
}

```

C++

```

int i;

for(i=0; i<4; i++){
    cout << ob[i][0].get_a() << ' ';
    cout << ob[i][1].get_a() << "\n";
}

cout << "\n";

```

This program displays -1 -2 -3 -4 on the screen. In this example, the values -1 through -4 are passed to the ob constructor function.

This program displays

```

1 2
3 4
5 6
7 8

```

4. As you know, a constructor can take more than one argument. When initializing an array of objects whose constructor takes more than one argument, you must use the alternative form of initialization mentioned earlier. Let's begin with an example:

```

#include <iostream>
using namespace std;

// Create a two-dimensional array of objects
#include <iostream>
using namespace std;

class samp {
    int a, b;
public:
    samp(int n, int m) { a = n; b = m; }
    int get_a() { return a; }
    int get_b() { return b; }
};

int main()
{
    samp ob[4][2] = {
        samp(1, 2), samp(3, 4),
        samp(5, 6), samp(7, 8),
        samp(9, 10), samp(11, 12),
        samp(13, 14), samp(15, 16)
    };
}

```

```

int i;
for(i=0; i<4; i++) {
    cout << ob[i][0].get_a() << ' ';
    cout << ob[i][0].get_b() << "\n";
    cout << ob[i][1].get_a() << ' ';
    cout << ob[i][1].get_b() << "\n";
}

cout << "\n";

return 0;
}

```

In this example, **samp**'s constructor takes two arguments. Here, the array **ob** is declared and initialized in **main()** by using direct calls to **samp**'s constructor. This is necessary because the formal C++ syntax allows only one argument at a time in a comma-separated list. There is no way, for example, to specify two (or more) arguments per entry in the list. Therefore, when you initialize arrays of objects that have constructors that take more than one argument, you must use the "long form" initialization syntax rather than the "shorthand form."

Note

You can always use the long form of initialization even if the object takes only one argument. It's just that the short form is more convenient in this case.

The preceding program displays

1 2

3 4

5 6

7 8

9 10

11 12

13 14

15 16

EXERCISES

1. Using the following class declaration, create a ten-element array and initialize the **ch** element with the values A through J. Demonstrate that the array does, indeed, contain these values.

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

class letters {
    char ch;
public:
    letters(char c) { ch = c; }
    char get_ch() { return ch; }
};
```

2. Using the following class declaration, create a ten-element array, initialize **num** to the values 1 through 10, and initialize **sqr** to **num**'s square.

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

class squares {
    int num, sqr;
public:
    squares(int a, int b) { num = a; sqr = b; }
    void show() { cout << num << ' ' << sqr << "\n"; }
};
```

3. Change the initialization in Exercise 1 so it uses the long form. (That is, invoke **letters**' constructor explicitly in the initialization list.)

4.2

USING POINTERS TO OBJECTS

As discussed in Chapter 2, objects can be accessed via pointers. As you know, when a pointer to an object is used, the object's members are referenced using the arrow (**->**) operator instead of the dot (**.**) operator.

Pointer arithmetic using an object pointer is the same as it is for any other data type: it is performed relative to the type of the object. For example, when an object pointer is incremented, it points to the next object. When an object pointer is decremented, it points to the previous object.

EXAMPLE

1. Here is an example of object pointer arithmetic:

```
// Pointers to objects.
#include <iostream>
using namespace std;

class samp {
    int a, b;
public:
    samp(int n, int m) { a = n; b = m; }
    int get_a() { return a; }
    int get_b() { return b; }
};

int main()
{
    samp ob[4] = {
        samp(1, 2),
        samp(3, 4),
        samp(5, 6),
        samp(7, 8)
    };
    int i;

    samp *p;
    p = ob; // get starting address of array

    for(i=0; i<4; i++) {
        cout << p->get_a() << ' ';
        cout << p->get_b() << "\n";
        p++; // advance to next object
    }

    cout << "\n";
}
```

C++

```
return 0;  
}
```

This program displays

```
1 2  
3 4  
5 6  
7 8
```

As evidenced by the output, each time **p** is incremented, it points to the next object in the array.

EXERCISES

1. Rewrite Example 1 so it displays the contents of the **ob** array in reverse order.
 2. Change Section 4.1, Example 3 so the two-dimensional array is accessed via a pointer. Hint: In C++, as in C, all arrays are stored contiguously, left to right, low to high.
-

4.3

THE **this** POINTER

C++ contains a special pointer that is called **this**. **this** is a pointer that is automatically passed to any member function when it is called, and it is a pointer to the object that generates the call. For example, given this statement,

```
ob.f1(); // assume that ob is an object
```

the function **f1()** is automatically passed a pointer to **ob**—which is the object that invokes the call. This pointer is referred to as **this**.

It is important to understand that only member functions are passed a **this** pointer. For example, a friend does not have a **this** pointer.

EXAMPLE

1. As you have seen, when a member function refers to another member of a class, it does so directly, without qualifying the member with either a class or an object specification. For example, examine this short program, which creates a simple inventory class:

```
// Demonstrate the this pointer.
#include <iostream>
#include <cstring>
using namespace std;

class inventory {
    char item[20];
    double cost;
    int on_hand;
public:
    inventory(char *i, double c, int o)
    {
        strcpy(item, i);
        cost = c;
        on_hand = o;
    }
    void show();
};

void inventory::show()
{
    cout << item;
    cout << ": $" << cost;
    cout << " On hand: " << on_hand << "\n";
}

int main()
{
    inventory ob("wrench", 4.95, 4);

    ob.show();

    return 0;
}
```

As you can see, within the constructor `inventory()` and the member function `show()`, the member variables `item`, `cost`,

and **on_hand** are referred to directly. This is because a member function can be called only in conjunction with an object. Therefore, the compiler knows which object's data is being referred to.

However, there is an even more subtle explanation. When a member function is called, it is automatically passed a **this** pointer to the object that invoked the call. Thus, the preceding program could be rewritten as shown here:

```
// Demonstrate the this pointer.
#include <iostream>
#include <cstring>
using namespace std;

class inventory {
    char item[20];
    double cost;
    int on_hand;
public:
    inventory(char *i, double c, int o)
    {
        strcpy(this->item, i); // access members
        this->cost = c; // through the this
        this->on_hand = o; // pointer
    }
    void show();
};

void inventory::show()
{
    cout << this->item; // use this to access members
    cout << ": $" << this->cost;
    cout << " On hand: " << this->on_hand << "\n";
}

int main()
{
    inventory ob("wrench", 4.95, 4);

    ob.show();

    return 0;
}
```


Here the member variables are accessed explicitly through the **this** pointer. Thus, within **show()**, these two statements are equivalent:

```
cost = 123.23;
this->cost = 123.23;
```

In fact, the first form is, loosely speaking, a shorthand for the second.

While no C++ programmer would use the **this** pointer to access a class member as just shown, because the shorthand form is much easier, it is important to understand what the shorthand implies.

The **this** pointer has several uses, including aiding in overloading operators. This use will be detailed in Chapter 6. For now, the important thing to understand is that by default, all member functions are automatically passed a pointer to the invoking object.

EXERCISE

1. Given the following program, convert all appropriate references to class members to explicit **this** pointer references.

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

class myclass {
    int a, b;
public:
    myclass(int n, int m) { a = n; b = m; }
    int add() { return a+b; }
    void show();
};

void myclass::show()
{
    int t;

    t = add(); // call member function
    cout << t << "\n";
}
```

C++

```
int main()
{
    myclass ob(10, 14);

    ob.show();

    return 0;
}
```

4.4

USING new AND delete

Up to now, when memory needed to be allocated, you have been using **malloc()**, and you have been freeing allocated memory by using **free()**. These are, of course, the standard C dynamic allocation functions. While these functions are available in C++, C++ provides a safer and more convenient way to allocate and free memory. In C++, you can allocate memory using **new** and release it using **delete**. These operators take these general forms:

```
p-var = new type;
delete p-var;
```

Here *type* is the type specifier of the object for which you want to allocate memory and *p-var* is a pointer to that type. **new** is an operator that returns a pointer to dynamically allocated memory that is large enough to hold an object of type *type*. **delete** releases that memory when it is no longer needed. **delete** can be called only with a pointer previously allocated with **new**. If you call **delete** with an invalid pointer, the allocation system will be destroyed, possibly crashing your program.

If there is insufficient available memory to fill an allocation request, one of two actions will occur. Either **new** will return a null pointer or it will generate an exception. (Exceptions and exception handling are described later in this book; loosely, an exception is a run-time error that can be managed in a structured fashion.) In Standard C++, the default behavior of **new** is to generate an exception when it cannot satisfy an allocation request. If this exception is not handled by your

program, your program will be terminated. The trouble is that the precise action that **new** takes on failure has been changed several times over the past few years. Thus, it is possible that your compiler will not implement **new** as defined by Standard C++.

When C++ was first invented, **new** returned null on failure. Later this was changed such that **new** caused an exception on failure. Finally, it was decided that a **new** failure will generate an exception by default, but that a null pointer could be returned instead, as an option. Thus, **new** has been implemented differently at different times by compiler manufacturers. For example, at the time of this writing, Microsoft's Visual C++ returns a null pointer when **new** fails. Borland C++ generates an exception. Although all compilers will eventually implement **new** in compliance with Standard C++, currently the only way to know the precise action of **new** on failure is to check your compiler's documentation.

Since there are two possible ways that **new** can indicate allocation failure, and since different compilers might do so differently, the code in this book will be written in such a way that both contingencies are accommodated. All code in this book will test the pointer returned by **new** for null. This handles compilers that implement **new** by returning null on failure, while causing no harm for those compilers for which **new** throws an exception. If your compiler generates an exception when **new** fails, the program will simply be terminated. Later, when exception handling is described, **new** will be re-examined and you will learn how to better handle an allocation failure. You will also learn about an alternative form of **new** that always returns a null pointer when an error occurs.

One last point: none of the examples in this book should cause **new** to fail, since only a handful of bytes are being allocated by any single program.

Although **new** and **delete** perform functions similar to **malloc()** and **free()**, they have several advantages. First, **new** automatically allocates enough memory to hold an object of the specified type. You do not need to use **sizeof**, for example, to compute the number of bytes required. This reduces the possibility for error. Second, **new** automatically returns a pointer of the specified type. You do not need to use an explicit type cast the way you did when you allocated memory by using **malloc()** (see the following note). Third, both **new** and **delete** can be overloaded, enabling you to easily implement your own custom allocation system. Fourth, it is possible to initialize a

dynamically allocated object. Finally, you no longer need to include `<cstdlib>` with your programs.

*In C, no type cast is required when you are assigning the return value of `malloc()` to a pointer because the `void *` returned by `malloc()` is automatically converted into a pointer compatible with the type of pointer on the left side of the assignment. However, this is not the case in C++, which requires an explicit type cast when you use `malloc()`. The reason for this difference is that it allows C++ to enforce more rigorous type checking.*

Now that `new` and `delete` have been introduced, they will be used instead of `malloc()` and `free()`.

EXAMPLES

1. As a short first example, this program allocates memory to hold an integer:

```
// A simple example of new and delete.
#include <iostream>
using namespace std;

int main()
{
    int *p;

    p = new int; // allocate room for an integer
    if(!p) {
        cout << "Allocation error\n";
        return 1;
    }

    *p = 1000;

    cout << "Here is integer at p: " << *p << "\n";

    delete p; // release memory

    return 0;
}
```

Notice that the value returned by **new** is checked before it is used. As explained earlier, this check is meaningful only if your compiler implements **new** in such a way that it returns null on failure.

2. Here is an example that allocates an object dynamically:

```
// Allocating dynamic objects.
#include <iostream>
using namespace std;

class samp {
    int i, j;
public:
    void set_ij(int a, int b) { i=a; j=b; }
    int get_product() { return i*j; }
};

int main()
{
    samp *p;

    p = new samp; // allocate object
    if(!p) {
        cout << "Allocation error\n";
        return 1;
    }

    p->set_ij(4, 5);

    cout << "Product is: " << p->get_product() << "\n";

    return 0;
}
```

EXERCISES

1. Write a program that uses **new** to dynamically allocate a **float**, a **long**, and a **char**. Give these dynamic variables values and display their values. Finally, release all dynamically allocated memory by using **delete**.

2. Create a class that contains a person's name and telephone number. Using **new**, dynamically allocate an object of this class and put your name and phone number into these fields within this object.
3. What are the two ways that **new** might indicate an allocation failure?

4.5**MORE ABOUT *new* AND *delete***

This section discusses two additional features of **new** and **delete**. First, dynamically allocated objects can be given initial values. Second, dynamically allocated arrays can be created.

You can give a dynamically allocated object an initial value by using this form of the **new** statement:

```
p-var = new type (initial-value);
```

To dynamically allocate a one-dimensional array, use this form of **new**:

```
p-var = new type [size];
```

After this statement has executed, *p-var* will point to the start of an array of *size* elements of the type specified. For various technical reasons, it is not possible to initialize an array that is dynamically allocated.

To delete a dynamically allocated array, use this form of **delete**:

```
delete [] p-var;
```

This syntax causes the compiler to call the destructor function for each element in the array. It does *not* cause *p-var* to be freed multiple times. *p-var* is still freed only once.

*For older compilers, you might need to specify the size of the array that you are deleting between the square brackets of the **delete** statement. This was required by the original definition of C++. However, the size specification is not needed by modern compilers.*

EXAMPLES

1. This program allocates memory for an integer and initializes that memory:

```
// An example of initializing a dynamic variable.
#include <iostream>
using namespace std;

int main()
{
    int *p;

    p = new int(9); // give initial value of 9

    if(!p) {
        cout << "Allocation error\n";
        return 1;
    }

    cout << "Here is integer at p: " << *p << "\n";

    delete p; // release memory

    return 0;
}
```

As you should expect, this program displays the value 9, which is the initial value given to the memory pointed to by **p**.

2. The following program passes initial values to a dynamically allocated object:

```
// Allocating dynamic objects.
#include <iostream>
using namespace std;

class samp {
    int i, j;
public:
    samp(int a, int b) { i=a; j=b; }
    int get_product() { return i*j; }
};

int main()
{
```

▼ C++

```
samp *p;

p = new samp(6, 5); // allocate object with initialization
if(!p) {
    cout << "Allocation error\n";
    return 1;
}

cout << "Product is: " << p->get_product() << "\n";

delete p;

return 0;
}
```

When the **samp** object is allocated, its constructor is automatically called and is passed the values 6 and 5.

3. The following program allocates an array of integers:

```
// A simple example of new and delete.
#include <iostream>
using namespace std;

int main()
{
    int *p;

    p = new int [5]; // allocate room for 5 integers

    // always make sure that allocation succeeded
    if(!p) {
        cout << "Allocation error\n";
        return 1;
    }

    int i;

    for(i=0; i<5; i++) p[i] = i;

    for(i=0; i<5; i++) {
        cout << "Here is integer at p[" << i << "]: ";
        cout << p[i] << "\n";
    }
}
```



```
delete [] p; // release memory

return 0;
}
```

This program displays the following:

```
Here is integer at p[0]: 0
Here is integer at p[1]: 1
Here is integer at p[2]: 2
Here is integer at p[3]: 3
Here is integer at p[4]: 4
```

4. The following program creates a dynamic array of objects:

```
// Allocating dynamic objects.
#include <iostream>
using namespace std;

class samp {
    int i, j;
public:
    void set_ij(int a, int b) { i=a; j=b; }
    int get_product() { return i*j; }
};

int main()
{
    samp *p;
    int i;

    p = new samp [10]; // allocate object array
    if(!p) {
        cout << "Allocation error\n";
        return 1;
    }

    for(i=0; i<10; i++)
        p[i].set_ij(i, i);

    for(i=0; i<10; i++) {
        cout << "Product [" << i << "] is: ";
        cout << p[i].get_product() << "\n";
    }
    delete [] p;
```

C++

```

return 0;
}

```

This program displays the following:

```

Product [0] is: 0
Product [1] is: 1
Product [2] is: 4
Product [3] is: 9
Product [4] is: 16
Product [5] is: 25
Product [6] is: 36
Product [7] is: 49
Product [8] is: 64
Product [9] is: 81

```

5. The following version of the preceding program gives **samp** a destructor, and now when **p** is freed, each element's destructor is called:

```

// Allocating dynamic objects.
#include <iostream>
using namespace std;

class samp {
    int i, j;
public:
    void set_ij(int a, int b) { i=a; j=b; }
    ~samp() { cout << "Destroying...\n"; }
    int get_product() { return i*j; }
};

int main()
{
    samp *p;
    int i;

    p = new samp [10]; // allocate object array
    if(!p) {
        cout << "Allocation error\n";
        return 1;
    }

    for(i=0; i<10; i++)
        p[i].set_ij(i, i);
}

```

```
for(i=0; i<10; i++) {
    cout << "Product [" << i << "] is: ";
    cout << p[i].get_product() << "\n";
}

delete [] p;
return 0;
}
```

This program displays the following:

```
Product [0] is: 0
Product [1] is: 1
Product [2] is: 4
Product [3] is: 9
Product [4] is: 16
Product [5] is: 25
Product [6] is: 36
Product [7] is: 49
Product [8] is: 64
Product [9] is: 81
Destroying...
Destroying...
Destroying...
Destroying...
Destroying...
Destroying...
Destroying...
Destroying...
Destroying...
Destroying...
```

As you can see, **samp**'s destructor is called ten times—once for each element in the array.

EXERCISES

1. Show how to convert the following code into its equivalent that uses **new**.

```
char *p;
```

```
p = (char *) malloc(100);
```

```
C++  
// ...  
strcpy(p, "This is a test");
```

Hint: A string is simply an array of characters.

- Using **new**, show how to allocate a **double** and give it an initial value of -123.0987.

4.6***R*EFERENCES**

C++ contains a feature that is related to the pointer: the *reference*. A reference is an implicit pointer that for all intents and purposes acts like another name for a variable. There are three ways that a reference can be used. First, a reference can be passed to a function. Second, a reference can be returned by a function. Finally, an independent reference can be created. Each of these applications of the reference is examined, beginning with reference parameters.

Without a doubt, the most important use of a reference is as a parameter to a function. To help you understand what a reference parameter is and how it works, let's first start with a program that uses a pointer (not a reference) as a parameter:

```
#include <iostream>  
using namespace std;  
  
void f(int *n); // use a pointer parameter  
  
int main()  
{  
    int i = 0;  
  
    f(&i);  
  
    cout << "Here is i's new value: " << i << '\n';
```

```

return 0;
}
void f(int *n)
{

```

```

    *n = 100; // put 100 into the argument pointed to by n
}

```

Here `f()` loads the value 100 into the integer pointed to by `n`. In this program, `f()` is called with the address of `i` in `main()`. Thus, after `f()` returns, `i` contains the value 100.

This program demonstrates how a pointer is used as a parameter to manually create a call-by-reference parameter-passing mechanism. In a C program, this is the only way to achieve a call-by-reference. However, in C++, you can completely automate this process by using a reference parameter. To see how, let's rework the previous program. Here is a version that uses a reference parameter:

```

#include <iostream>
using namespace std;

```

```

void f(int &n); // declare a reference parameter

```

```

int main()
{

```

```

    int i = 0;

```

```

    f(i);

```

```

    cout << "Here is i's new value: " << i << "\n";

```

```

    return 0;
}

```

```

// f() now uses a reference parameter

```

```

void f(int &n)
{

```

```

    // notice that no * is needed in the following statement

```

```

    n = 100; // put 100 into the argument used to call f()
}

```

Examine this program carefully. First, to declare a reference variable or parameter, you precede the variable's name with the `&`. This is how `n` is declared as a parameter to `f()`. Now that `n` is a

C++

reference, it is no longer necessary—or even legal—to apply the * operator. Instead, each time **n** is used within **f()**, it is automatically treated as a pointer to the argument used to call **f()**. This means that the statement

```
n = 100;
```

actually puts the value 100 into the variable used to call **f()**, which, in this case, is **i**. Further, when **f()** is called, there is no need to precede the argument with the **&**. Instead, because **f()** is declared as taking a reference parameter, the address to the argument is *automatically* passed to **f()**.

To review, when you use a reference parameter, the compiler automatically passes the address of the variable used as the argument. There is no need to manually generate the address of the argument by preceding it with an **&** (in fact, it is not allowed). Further, within the function, the compiler automatically uses the variable pointed to by the reference parameter. There is no need to employ the * (and again, it is not allowed). Thus, a reference parameter fully automates the call-by-reference parameter-passing mechanism.

It is important to understand that you cannot change what a reference is pointing to. For example, if the statement

```
n++;
```

were put inside **f()** in the preceding program, **n** would still be pointing to **i** in **main()**. Instead of incrementing **n**, this statement increments the value of the variable being referenced (in this case, **i**).

Reference parameters offer several advantages over their (more or less) equivalent pointer alternatives. First, from a practical point of view, you no longer need to remember to pass the address of an argument. When a reference parameter is used, the address is automatically passed. Second, in the opinion of many programmers, reference parameters offer a cleaner, more elegant interface than the rather clumsy explicit pointer mechanism. Third, as you will see in the next section, when an object is passed to a function as a reference, no copy is made. This is one way to eliminate the troubles associated with the copy of an argument damaging something needed elsewhere.

when its destructor function is called.

EXAMPLES

1. The classic example of passing arguments by reference is a function that exchanges the values of the two arguments with which it is called. Here is an example called `swapargs()` that uses references to swap its two integer arguments:

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

```
void swapargs(int &x, int &y);
```

```
int main() (Hafsa, for 24)
```

```
{
```

```
    int i, j;
```

```
    i = 10;
```

```
    j = 19;
```

```
    cout << "i: " << i << ", ";
```

```
    cout << "j: " << j << "\n";
```

```
    swapargs(i, j);
```

```
    cout << "After swapping: ";
```

```
    cout << "i: " << i << ", ";
```

```
    cout << "j: " << j << "\n";
```

```
    return 0;
```

```
}
```

```
void swapargs(int &x, int &y)
```

```
{
```

```
    int t;
```

```
    t = x;
```

```
    x = y;
```

```
    y = t;
```

```
}
```

If `swapargs()` had been written using pointers instead of references, it would have looked like this:

C++

```

void swapargs(int *x, int *y)
{
    int t;

    t = *x;
    *x = *y;
    *y = t;
}

```

As you can see, by using the reference version of `swapargs()`, the need for the `*` operator is eliminated.

2. Here is a program that uses the `round()` function to round a **double** value. The value to be rounded is passed by reference.

```

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

```

```

void round(double &num);

```

```

int main()
{
    double i = 100.4;

    cout << i << " rounded is ";
    round(i);
    cout << i << "\n";

    i = 10.9;
    cout << i << " rounded is ";
    round(i);
    cout << i << "\n";

    return 0;
}

```

```

void round(double &num)
{

```

```

    double frac;
    double val;

```

```

    // decompose num into whole and fractional parts
    frac = modf(num, &val);

```

```

    if(frac < 0.5) num = val;

```



```
else num = val+1.0;
}
```

round() uses a relatively obscure standard library function called **modf()** to decompose a number into its whole number and fractional parts. The fractional part is returned; the whole number is put into the variable pointed to by **modf()**'s second parameter.

EXERCISES

1. Write a function called **neg()** that reverses the sign of its integer parameter. Write the function two ways—first by using a pointer parameter and then by using a reference parameter. Include a short program to demonstrate their operation.
2. What is wrong with the following program?

```
// This program has an error.
```

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

```
void triple(double &num);
```

```
int main()
```

```
{
    double d = 7.0;
```

```
    triple(&d);
```

```
    cout << d;
```

```
    return 0;
```

```
}
```

```
// Triple num's value.
```

```
void triple(double &num)
```

```
{
    num = 3 * num;
```

```
}
```

3. Give some advantages of reference parameters.

***P*ASSING REFERENCES TO OBJECTS**

As you learned in Chapter 3, when an object is passed to a function by use of the default call-by-value parameter-passing mechanism, a copy of that object is made. Although the parameter's constructor function is not called, its destructor function is called when the function returns. As you should recall, this can cause serious problems in some instances—when the destructor frees dynamic memory, for example.

One solution to this problem is to pass an object by reference. (The other solution involves the use of copy constructors, which are discussed in Chapter 5.) When you pass the object by reference, no copy is made, and therefore its destructor function is not called when the function returns. Remember, however, that changes made to the object inside the function affect the object used as the argument.

It is critical to understand that a reference is not a pointer. Therefore, when an object is passed by reference, the member access operator remains the dot (.), not the arrow (->).

EXAMPLE

1. The following is an example that demonstrates the usefulness of passing an object by reference. First, here is a version of a program that passes an object of **myclass** by value to a function called **f()**:

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

class myclass {
    int who;
public:
    myclass(int n) {
        who = n;
        cout << "Constructing " << who << "\n";
    }
}
```

```

~myclass() { cout << "Destructing " << who << "\n"; }
int id() { return who; }
};

// o is passed by value.
void f(myclass o)
{
    cout << "Received " << o.id() << "\n";
}

int main()
{
    myclass x(1);

    f(x);

    return 0;
}

```

This function displays the following:

```

Constructing 1
Received 1
Destructing 1
Destructing 1

```

As you can see, the destructor function is called twice—first when the copy of object 1 is destroyed when `f()` terminates and again when the program finishes.

However, if the program is changed so that `f()` uses a reference parameter, no copy is made and, therefore, no destructor is called when `f()` returns:

```

#include <iostream>
using namespace std;

class myclass {
    int who;
public:
    myclass(int n) {
        who = n;
        cout << "Constructing " << who << "\n";
    }
    ~myclass() { cout << "Destructing " << who << "\n"; }
    int id() { return who; }
}

```

```
};

// Now o is passed by reference.
void f(myclass &o)
{
    // note that . operator is still used!!!
    cout << "Received " << o.id() << "\n";
}

int main()
{
    myclass x(1);

    f(x);

    return 0;
}
```

This version displays the following output:

```
Constructing 1
Received 1
Destructing 1
```

**Remember**

When accessing members of an object by using a reference, use the dot operator, not the arrow.

EXERCISE

1. What is wrong with the following program? Show how it can be fixed by using a reference parameter.

```
// This program has an error.
#include <iostream>
#include <cstring>
#include <cstdlib>
using namespace std;

class strtype {
    char *p;
public:
    strtype(char *s);
```

```
~strtype() { delete [] p; }
char *get() { return p; }
};

strtype::strtype(char *s)
{
    int l;

    l = strlen(s)+1;

    p = new char [l];
    if(!p) {
        cout << "Allocation error\n";
        exit(1);
    }

    strcpy(p, s);
}

void show(strtype x)
{
    char *s;

    s = x.get();
    cout << s << "\n";
}

int main()
{
    strtype a("Hello"), b("There");

    show(a);
    show(b);

    return 0;
}
```

4.8

R RETURNING REFERENCES

A function can return a reference. As you will see in Chapter 6, returning a reference can be very useful when you are overloading

certain types of operators. However, it also can be employed to allow a function to be used on the left side of an assignment statement. The effect of this is both powerful and startling.

EXAMPLES

1. To begin, here is a very simple program that contains a function that returns a reference:

```
// A simple example of a function returning a reference.
#include <iostream>
using namespace std;

int &f(); // return a reference
int x;

int main()
{
    f() = 100; // assign 100 to reference returned by f()

    cout << x << "\n";

    return 0;
}

// Return an int reference.
int &f()
{
    return x; // returns a reference to x
}
```

Here function **f()** is declared as returning a reference to an integer. Inside the body of the function, the statement

```
return x;
```

does *not* return the value of the global variable **x**, but rather, it automatically returns **x**'s address (in the form of a reference).

Thus, inside **main()**, the statement

```
f() = 100;
```

puts the value 100 into **x** because **f()** has returned a reference to it.

To review, function **f()** returns a reference. Thus, when **f()** is used on the left side of the assignment statement, it is this

reference, returned by `f()`, that is being assigned to. Since `f()` returns a reference to `x` (in this example), it is `x` that receives the value 100.

2. You must be careful when returning a reference that the object you refer to does not go out of scope. For example, consider this slight reworking of function `f()`:

```
// Return an int reference.
int &f()
{
    int x; // x is now a local variable
    return x; // returns a reference to x
}
```

In this case, `x` is now local to `f()` and will go out of scope when `f()` returns. This effectively means that the reference returned by `f()` is useless.

**Note**

Some C++ compilers will not allow you to return a reference to a local variable. However, this type of problem can manifest itself in other ways, such as when objects are allocated dynamically.

3. One very good use of returning a reference is found when a bounded array type is created. As you know, in C and C++, no array boundary checking occurs. It is therefore possible to overflow or underflow an array. However, in C++, you can create an array class that performs automatic bounds checking. An array class contains two core functions—one that stores information into the array and one that retrieves information. These functions can check, at run time, that the array boundaries are not overrun.

The following program implements a bounds-checking array for characters:

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

class array {
    int size;
```

```
    char *p;
public:
    array(int num);
    ~array() { delete [] p; }
    char &put(int i);
    char get(int i);
};

array::array(int num)
{
    p = new char [num];
    if(!p) {
        cout << "Allocation error\n";
        exit(1);
    }
    size = num;
}

// Put something into the array.
char &array::put(int i)
{
    if(i<0 || i>=size) {
        cout << "Bounds error!!!\n";
        exit(1);
    }
    return p[i]; // return reference to p[i]
}

// Get something from the array.
char array::get(int i)
{
    if(i<0 || i>=size) {
        cout << "Bounds error!!!\n";
        exit(1);
    }
    return p[i]; // return character
}

int main()
{
    array a(10);
```



```
a.put(3) = 'X';
a.put(2) = 'R';

cout << a.get(3) << a.get(2);
cout << "\n";

// now generate run-time boundary error
a.put(11) = '!';

return 0;
}
```

This example is a practical use of functions returning references, and you should examine it closely. Notice that the **put()** function returns a reference to the array element specified by parameter **i**. This reference can then be used on the left side of an assignment statement to store something in the array—if the index specified by **i** is not out of bounds. The reverse is **get()**, which returns the value stored at the specified index if that index is within range. This approach to maintaining an array is sometimes referred to as a *safe array*. (You will see a better way to create a safe array later on, in Chapter 6.)

One other thing to notice about the preceding program is that the array is allocated dynamically by the use of **new**. This allows arrays of differing lengths to be declared.

As mentioned, the way that bounds checking is performed in this program is a practical application of C++. If you need to have array boundaries verified at run time, this is one way to do it. However, remember that bounds checking slows access to the array. Therefore, it is best to include bounds checking only when there is a real likelihood that an array boundary will be violated.

EXERCISES

1. Write a program that creates a two-by-three two-dimensional safe array of integers. Demonstrate that it works.
2. Is the following fragment valid? If not, why not?

```
int &f();  
.  
.  
.  
int *x;  
  
x = f();
```

INDEPENDENT REFERENCES AND RESTRICTIONS

Although not commonly used, the *independent reference* is another type of reference that is available in C++. An independent reference is a reference variable that in all effects is simply another name for another variable. Because references cannot be assigned new values, an independent reference must be initialized when it is declared.

Because independent references are sometimes used, it is important that you know about them. However, most programmers feel that there is no need for them and that they can add confusion to a program. Further, independent references exist in C++ largely because there was no compelling reason to disallow them. But for the most part, their use should be avoided.

There are a number of restrictions that apply to all types of references. You cannot reference another reference. You cannot obtain the address of a reference. You cannot create arrays of references, and you cannot reference a bit-field. References must be initialized unless they are members of a class, are return values, or are function parameters.

**Remember**

References are similar to pointers, but they are not pointers.

EXAMPLES

1. Here is a program that contains an independent reference:

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

int main()
{
    int x;
    int &ref = x; // create an independent reference

    x = 10; // these two statements
    ref = 10; // are functionally equivalent

    ref = 100;
    // this prints the number 100 twice
    cout << x << ' ' << ref << "\n";

    return 0;
}
```

In this program, the independent reference **ref** serves as a different name for **x**. From a practical point of view, **x** and **ref** are equivalent.

2. An independent reference can refer to a constant. For example, this is valid:

```
const int &ref = 10;
```

Again, there is little benefit in this type of reference, but you may see it from time to time in other programs.

EXERCISE

1. On your own, try to think of a good use for an independent reference.

SKILLS CHECK**Mastery
Skills Check**

At this point, you should be able to perform the following exercises and answer the questions.

1. Given the following class, create a two-by-five two-dimensional array and give each object in the array an initial value of your own choosing. Then display the contents of the array.

```
class a_type {  
    double a, b;  
public:  
    a_type(double x, double y) {  
        a = x;  
        b = y;  
    }  
    void show() { cout << a << ' ' << b << "\n"; }  
};
```

2. Modify your solution to the preceding problem so it accesses the array by using a pointer.
3. What is the **this** pointer?
4. Show the general forms for **new** and **delete**. What are some advantages of using them instead of **malloc()** and **free()**?
5. What is a reference? What is one advantage of using a reference parameter?

6. Create a function called `recip()` that takes one **double** reference parameter. Have the function change the value of that parameter into its reciprocal. Write a program to demonstrate that it works.



Cumulative
Skills Check

This section checks how well you have integrated material in this chapter with that from the preceding chapters.

1. Given a pointer to an object, what operator is used to access a member of that object?
2. In Chapter 2, a **strtype** class was created that dynamically allocated space for a string. Rework the **strtype** class (shown here for your convenience) so it uses **new** and **delete**.

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

class strtype {
    char *p;
    int len;
public:
    strtype(char *ptr);
    ~strtype();
    void show();
};

strtype::strtype(char *ptr)
{
    len = strlen(ptr);
    p = (char *) malloc(len+1);
    if(!p) {
        cout << "Allocation error\n";
        exit(1);
    }
    strcpy(p, ptr);
}
```



```
strtype::~strtype()
{
    cout << "Freeing p\n";
    free(p);
}

void strtype::show()
{
    cout << p << " - length: " << len;
    cout << "\n";
}

int main()
{
    strtype s1("This is a test."), s2("I like C++.");

    s1.show();
    s2.show();

    return 0;
}
```

3. On your own, rework any program from the preceding chapter so that it uses a reference.



5

Function Overloading

chapter objectives

- 5.1 Overloading constructor functions
- 5.2 Creating and using a copy constructor
- 5.3 The **overload** anachronism
- 5.4 Using default arguments
- 5.5 Overloading and ambiguity
- 5.6 Finding the address of an overloaded function



IN this chapter you will learn more about overloading functions. Although this topic was introduced early in this book, there are several further aspects of it that need to be covered. Among the topics included are how to overload constructor functions, how to create a copy constructor, how to give functions default arguments, and how to avoid ambiguity when overloading.



Before proceeding, you should be able to correctly answer the following questions and do the exercises.

1. What is a reference? Give two important uses.
2. Show how to allocate a **float** and an **int** by using **new**. Also, show how to free them by using **delete**.
3. What is the general form of **new** that is used to initialize a dynamic variable? Give a concrete example.
4. Given the following class, show how to initialize a ten-element array so that **x** has the values 1 through 10.

```
class samp {  
    int x;  
public:  
    samp(int n) { x = n; }  
    int getx() { return x; }  
};
```

5. Give one advantage of reference parameters. Give one disadvantage.
6. Can dynamically allocated arrays be initialized?
7. Create a function called **mag()** using the following prototype that raises **num** to the order of magnitude specified by **order**:

```
void mag(long &num, long order);
```


For example, if **num** is 4 and order is 2, when **mag()** returns, **num** will be 400. Demonstrate in a program that the function works.

5.1

OVERLOADING CONSTRUCTOR FUNCTIONS

It is possible—indeed, common—to overload a class's constructor function. (It is not possible to overload a destructor, however.) There are three main reasons why you will want to overload a constructor function: to gain flexibility, to support arrays, and to create copy constructors. The first two of these are discussed in this section. Copy constructors are discussed in the next section.

One thing to keep in mind as you study the examples is that there must be a constructor function for each way that an object of a class will be created. If a program attempts to create an object for which no matching constructor is found, a compile-time error occurs. This is why overloaded constructor functions are so common to C++ programs.

EXAMPLES

1. Perhaps the most frequent use of overloaded constructor functions is to provide the option of either giving an object an initialization or not giving it one. For example, in the following program, **o1** is given an initial value, but **o2** is not. If you remove the constructor that has the empty argument list, the program will not compile because there is no constructor that matches a noninitialized object of type **samp**. The reverse is also true: If you remove the parameterized constructor, the program will not compile because there is no match for an initialized object. Both are needed for this program to compile correctly.

C++

```

#include <iostream>
using namespace std;

class myclass {
    int x;
public:
    // overload constructor two ways
    myclass() { x = 0; } // no initializer
    myclass(int n) { x = n; } // initializer
    int getx() { return x; }
};

int main()
{
    myclass o1(10); // declare with initial value
    myclass o2; // declare without initializer

    cout << "o1: " << o1.getx() << '\n';
    cout << "o2: " << o2.getx() << '\n';

    return 0;
}

```

2. Another common reason constructor functions are overloaded is to allow both individual objects and arrays of objects to occur within a program. As you probably know from your own programming experience, it is fairly common to initialize a single variable, but it is not as common to initialize an array. (Quite often array values are assigned using information known only when the program is executing.) Thus, to allow noninitialized arrays of objects along with initialized objects, you must include a constructor that supports initialization and one that does not.

For instance, assuming the class **myclass** from Example 1, both of these declarations are valid:

```

myclass ob(10);
myclass ob[5];

```

By providing both a parameterized and a parameterless constructor, your program allows the creation of objects that are either initialized or not as needed.

Of course, once you have defined both parameterized and parameterless constructors you can use them to create

initialized and noninitialized arrays. For example, the following program declares two arrays of type **myclass**; one is initialized and the other is not:

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

class myclass {
    int x;
public:
    // overload constructor two ways
    myclass() { x = 0; } // no initializer
    myclass(int n) { x = n; } // initializer
    int getx() { return x; }
};

int main()
{
    myclass o1[10]; // declare array without initializers

    // declare with initializers
    myclass o2[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

    int i;

    for(i=0; i<10; i++) {
        cout << "o1[" << i << "]: " << o1[i].getx();
        cout << '\n';
        cout << "o2[" << i << "]: " << o2[i].getx();
        cout << '\n';
    }

    return 0;
}
```

In this example, all elements of **o1** are set to 0 by the constructor function. The elements of **o2** are initialized as shown in the program.

- Another reason for overloading constructor functions is to allow the programmer to select the most convenient method of initializing an object. To see how, first examine the next example, which creates a class that holds a calendar date. It overloads the **date()** constructor two ways. One form accepts

the date as a character string. In the other form, the date is passed as three integers.

```
#include <iostream>
#include <cstdio> // included for sscanf()
using namespace std;

class date {
    int day, month, year;
public:
    date(char *str);
    date (int m, int d, int y) {
        day = d;
        month = m;
        year = y;
    }
    void show() {
        cout << month << '/' << day << '/';
        cout << year << '\n';
    }
};

date::date(char *str)
{
    sscanf(str, "%d%c%d%c%d", &month, &day, &year);
}

int main()
{
    // construct date object using string
    date sdate("12/31/99");

    // construct date object using integers
    date idate(12, 31, 99);

    sdate.show();
    idate.show();
    return 0;
}
```

The advantage of overloading the `date()` constructor, as shown in this program, is that you are free to use whichever version most conveniently fits the situation in which it is being used. For example, if a `date` object is being created from user input, the string version is the easiest to use. However, if the `date`

object is being constructed through some sort of internal computation, the three-integer parameter version probably makes more sense.

Although it is possible to overload a constructor as many times as you want, doing so excessively has a destructuring effect on the class. From a stylistic point of view, it is best to overload a constructor to accommodate only those situations that are likely to occur frequently. For example, overloading `date()` a third time so the date can be entered in terms of milliseconds makes little sense. However, overloading it to accept an object of type `time_t` (a type that stores the system date and time) could be very valuable. (See the Mastery Skills Check exercises at the end of this chapter for an example that does just this.)

- ④ There is one other situation in which you will need to overload a class's constructor function: when a dynamic array of that class will be allocated. As you should recall from the preceding chapter, a dynamic array cannot be initialized. Thus, if the class contains a constructor that takes an initializer, you must include an overloaded version that takes no initializer. For example, here is a program that allocates an object array dynamically:

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

class myclass {
    int x;
public:
    // overload constructor two ways
    myclass() { x = 0; } // no initializer
    myclass(int n) { x = n; } // initializer
    int getx() { return x; }
    void setx(int n) { x = n; }
};

int main()
{
    myclass *p;
    myclass ob(10); // initialize single variable

    p = new myclass[10]; // can't use initializers here
    if(!p) {
```

C++

```

    cout << "Allocation error\n";
    return 1;
}

int i;

// initialize all elements to ob
for(i=0; i<10; i++) p[i] = ob;

for(i=0; i<10; i++) {
    cout << "p[" << i << "]: " << p[i].getx();
    cout << '\n';
}

return 0;
}

```

Without the overloaded version of `myclass()` that has no initializer, the `new` statement would have generated a compile-time error and the program would not have been compiled.

EXERCISES

1. Given this partially defined class

```

class strtype {
    char *p;
    int len;
public:
    char *getstring() { return p; }
    int getlength() { return len; }
};

```

add two constructor functions. Have the first one take no parameters. Have this one allocate 255 bytes of memory (using `new`), initialize that memory as a null string, and give `len` a value of 255. Have the other constructor take two parameters. The first is the string to use for initialization and the other is the number of bytes to allocate. Have this version allocate the specified amount of memory and copy the string to that

memory. Perform all necessary boundary checks and demonstrate that your constructors work by including a short program.

2. In Exercise 2 of Chapter 2, Section 2.1, you created a stopwatch emulation. Expand your solution so that the **stopwatch** class provides both a parameterless constructor (as it does already) and an overloaded version that accepts the system time in the form returned by the standard function **clock()**. Demonstrate that your improvement works.
 3. On your own, think about ways in which an overloaded constructor function can be beneficial to your own programming tasks.
-

CREATING AND USING A COPY CONSTRUCTOR

One of the more important forms of an overloaded constructor is the *copy constructor*. As numerous examples from the preceding chapters have shown, problems can occur when an object is passed to or returned from a function. As you will learn in this section, one way to avoid these problems is to define a copy constructor.

To begin, let's restate the problem that a copy constructor is designed to solve. When an object is passed to a function, a bitwise (i.e., exact) copy of that object is made and given to the function parameter that receives the object. However, there are cases in which this identical copy is not desirable. For example, if the object contains a pointer to allocated memory, the copy will point to the *same* memory as does the original object. Therefore, if the copy makes a change to the contents of this memory, it will be changed for the original object too! Also, when the function terminates, the copy will be destroyed, causing its destructor to be called. This might lead to undesired side effects that further affect the original object.



Copy constructors do not affect assignment operations.

A similar situation occurs when an object is returned by a function. The compiler will commonly generate a temporary object that holds a copy of the value returned by the function. (This is done automatically and is beyond your control.) This temporary object goes out of scope once the value is returned to the calling routine, causing the temporary object's destructor to be called. However, if the destructor destroys something needed by the calling routine (for example, if it frees dynamically allocated memory), trouble will follow.

At the core of these problems is the fact that a bitwise copy of the object is being made. To prevent these problems, you, the programmer, need to define precisely what occurs when a copy of an object is made so that you can avoid undesired side effects. The way you accomplish this is by creating a copy constructor. By defining a copy constructor, you can fully specify exactly what occurs when a copy of an object is made.

It is important for you to understand that C++ defines two distinct types of situations in which the value of one object is given to another. The first situation is assignment. The second situation is initialization, which can occur three ways:

- ▼ when an object is used to initialize another in a declaration statement,
- ▼ when an object is passed as a parameter to a function, and
- ▼ when a temporary object is created for use as a return value by a function.

The copy constructor only applies to initializations. It does not apply to assignments.

By default, when an initialization occurs, the compiler will automatically provide a bitwise copy. (That is, C++ automatically provides a default copy constructor that simply duplicates the object.) However, it is possible to specify precisely how one object will initialize another by defining a copy constructor. Once defined, the copy constructor is called whenever an object is used to initialize another.

The most common form of copy constructor is shown here:

```
classname (const classname &obj) {
    // body of constructor
}
```

Here *obj* is a reference to an object that is being used to initialize another object. For example, assuming a class called **myclass**, and that **y** is an object of type **myclass**, the following statements would invoke the **myclass** copy constructor:

```
myclass x = y; // y explicitly initializing x
func1(y);     // y passed as a parameter
y = func2();  // y receiving a returned object
```

In the first two cases, a reference to **y** would be passed to the copy constructor. In the third, a reference to the object returned by **func2()** is passed to the copy constructor.

EXAMPLES

1. Here is an example that illustrates why an explicit copy constructor function is needed. This program creates a very limited "safe" integer array type that prevents array boundaries from being overrun. Storage for each array is allocated using **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 dynamically allocated, a copy constructor
   is provided to allocate memory when one array object is
   used to initialize another.
```

```
*/
```

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

```
class array {
    int *p;
    int size;
public:
    array(int sz) { // constructor
```

```
p = new int[sz];
if(!p) exit(1);
size = sz;
cout << "Using 'normal' constructor\n";
}
~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.
```

In the following, memory is allocated specifically for the copy, and the address of this memory is assigned to p. Therefore, p is not pointing to the same dynamically allocated memory as the original object.

```
*/
array::array(const array &a) {
    int i;

    size = a.size;
    p = new int[a.size]; // allocate memory for copy
    if(!p) exit(1);
    for(i=0; i<a.size; i++) p[i] = a.p[i]; // copy contents
    cout << "Using copy constructor\n";
}

int main()
{
    array num(10); // this calls "normal" constructor
    int i;

    // put some values into the array
    for(i=0; i<10; i++) num.put(i, i);
}
```

```

// display num
for(i=9; i>=0; i--) cout << num.get(i);
cout << "\n";

// create another array and initialize with num
array x = num; // this invokes copy constructor

// display x
for(i=0; i<10; i++) cout << x.get(i);

return 0;
}

```

When **num** is used to initialize **x**, 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 have 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 bitwise initialization **array x = num** 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.)

The copy constructor is called only for initializations. For example, the following 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.

2. To see how the copy constructor helps prevent some of the problems associated with passing certain types of objects to functions, consider this (incorrect) program:

```

// This program has an error.
#include <iostream>
#include <cstring>
#include <cstdlib>
using namespace std;

```

C++

```
class strtype {
    char *p;
public:
    strtype(char *s);
    ~strtype() { delete [] p; }
    char *get() { return p; }
};

strtype::strtype(char *s)
{
    int l;

    l = strlen(s)+1;

    p = new char [l];
    if(!p) {
        cout << "Allocation error\n";
        exit(1);
    }
    strcpy(p, s);
}

void show(strtype x)
{
    char *s;

    s = x.get();
    cout << s << "\n";
}

int main()
{
    strtype a("Hello"), b("There");

    show(a);
    show(b);

    return 0;
}
```

In this program, when a **strtype** object is passed to **show()**, a bitwise copy is made (since no copy constructor has been defined) and put into parameter **x**. Thus, when the function

returns, **x** goes out of scope and is destroyed. This, of course, causes **x**'s destructor to be called, which frees **x.p**. However, the memory being freed is the same memory that is still being used by the object used to call the function. This results in an error.

The solution to the preceding problem is to define a copy constructor for the **strtype** class that allocates memory for the copy when the copy is created. This approach is used by the following, corrected, program:

```

/* This program uses a copy constructor to allow strtype objects
   to be passed to functions. */
#include <iostream>
#include <cstring>
#include <cstdlib>
using namespace std;

class strtype {
    char *p;
public:
    strtype(char *s); // constructor
    strtype(const strtype &o); // copy constructor
    ~strtype() { delete [] p; } // destructor
    char *get() { return p; }
};

// "Normal" constructor
strtype::strtype(char *s)
{
    int l;

    l = strlen(s)+1;
    p = new char [l];
    if(!p) {
        cout << "Allocation error\n";
        exit(1);
    }

    strcpy(p, s);
}

// Copy constructor
strtype::strtype(const strtype &o)
{

```

```
    C++
    int l;

    l = strlen(o.p)+1;

    p = new char [l]; // allocate memory for new copy
    if(!p) {
        cout << "Allocation error\n";
        exit(1);
    }

    strcpy(p, o.p); // copy string into copy
}

void show(strtype x)
{
    char *s;

    s = x.get();
    cout << s << "\n";
}

int main()
{
    strtype a("Hello"), b("There");

    show(a);
    show(b);

    return 0;
}
```

Now when **show()** terminates and **x** goes out of scope, the memory pointed to by **x.p** (which will be freed) is not the same as the memory still in use by the object passed to the function.

EXERCISES

1. The copy constructor is also invoked when a function generates the temporary object that is used as the function's return value (for those functions that return objects). With this in mind, consider the following output:

Constructing normally
Constructing normally
Constructing copy

This output was created by the following program. Explain why, and describe precisely what is occurring.

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

class myclass {
public:
    myclass();
    myclass(const myclass &o);
    myclass f();
};

// Normal constructor
myclass::myclass()
{
    cout << "Constructing normally\n";
}

// Copy constructor
myclass::myclass(const myclass &o)
{
    cout << "Constructing copy\n";
}

// Return an object.
myclass myclass::f()
{
    myclass temp;

    return temp;
}

int main()
{
    myclass obj;

    obj = obj.f();

    return 0;
}
```

2. Explain what is wrong with the following program and then fix it.

```
// This program contains an error.
#include <iostream>
#include <cstdlib>
using namespace std;

class myclass {
    int *p;
public:
    myclass(int i);
    ~myclass() { delete p; }
    friend int getval(myclass o);
};

myclass::myclass(int i)
{
    p = new int;

    if(!p) {
        cout << "Allocation error\n";
        exit(1);
    }
    *p = i;
}

int getval(myclass o)
{
    return *o.p; // get value
}

int main()
{
    myclass a(1), b(2);

    cout << getval(a) << " " << getval(b);
    cout << "\n";
    cout << getval(a) << " " << getval(b);

    return 0;
}
```

3. In your own words, explain the purpose of a copy constructor and how it differs from a normal constructor.

This

5.3

THE OVERLOAD ANACHRONISM

When C++ was first invented, the keyword **overload** was required to create an overloaded function. Although **overload** is now obsolete and no longer supported by modern C++ compilers, you may still see **overload** used in old programs, so it is a good idea to understand how it was applied.

The general form of **overload** is shown here,

```
overload func-name;
```

where *func-name* is the name of the function to be overloaded. This statement must precede the overloaded function declarations. For example, this tells the compiler that you will be overloading a function called **timer()**:

```
overload timer;
```

overload is obsolete and no longer supported by modern C++ compilers.

USING DEFAULT ARGUMENTS

There is a feature of C++ that is related to function overloading. This feature is called the *default argument*, and it allows you to give a parameter a default value when no corresponding argument is specified when the function is called. As you will see, using default arguments is essentially a shorthand form of function overloading.

To give a parameter a default argument, simply follow that parameter with an equal sign and the value you want it to default to if no corresponding argument is present when the function is called. For example, this function gives its two parameters default values of 0:

```
void f(int a=0, int b=0);
```

Notice that this syntax is similar to variable initialization. This function can now be called three different ways. First, it can be called with both arguments specified. Second, it can be called with only the first argument specified. In this case, **b** will default to 0. Finally, **f()** can be

called with no arguments, causing both **a** and **b** to default to 0. That is, the following invocations of **f()** are all valid:

```
f(); // a and b default to 0
f(10); // a is 10, b defaults to 0
f(10, 99) // a is 10, b is 99
```

In this example, it should be clear that there is no way to default **a** and specify **b**.

When you create a function that has one or more default arguments, those arguments must be specified only once: either in the function's prototype or in its definition if the definition precedes the function's first use. The defaults cannot be specified in both the prototype and the definition. This rule applies even if you simply duplicate the same defaults.

As you can probably guess, all default parameters must be to the right of any parameters that don't have defaults. Further, once you begin to define default parameters, you cannot specify any parameters that have no defaults.

One other point about default arguments: they must be constants or global variables. They cannot be local variables or other parameters.

EXAMPLES

1. Here is a program that illustrates the example described in the preceding discussion:

```
// A simple first example of default arguments.
#include <iostream>
using namespace std;

void f(int a=0, int b=0)
{
    cout << "a: " << a << ", b: " << b;
    cout << '\n';
}

int main()
{
    f();
    f(10);
    f(10, 99);
}
```

```

    return 0;
}

```

As you should expect, this program displays the following output:

```

a: 0, b: 0
a: 10, b: 0
a: 10, b: 99

```

Remember that once the first default argument is specified, all following parameters must have defaults as well. For example, this slightly different version of `f()` causes a compile-time error:

```

void f(int a=0, int b) // wrong! b must have default, too
{
    cout << "a: " << a << ", b: " << b;
    cout << '\n';
}

```

2. To understand how default arguments are related to function overloading, first consider the next program, which overloads the function called `rect_area()`. This function returns the area of a rectangle.

```

// Compute area of a rectangle using overloaded functions.
#include <iostream>
using namespace std;

```

```

// Return area of a non-square rectangle.
double rect_area(double length, double width)
{
    return length * width;
}

```

```

// Return area of a square.
double rect_area(double length)
{
    return length * length;
}

```

```

int main()
{
    cout << "10 x 5.8 rectangle has area: ";
    cout << rect_area(10.0, 5.8) << '\n';
}

```

```

    cout << "10 x 10 square has area: ";
    cout << rect_area(10.0) << '\n';
    return 0;
}

```

In this program, `rect_area()` is overloaded two ways. In the first way, both dimensions of a rectangle are passed to the function. This version is used when the rectangle is not a square. However, when the rectangle is a square, only one argument need be specified, and the second version of `rect_area()` is called.

If you think about it, it is clear that in this situation there is really no need to have two different functions. Instead, the second parameter can be defaulted to some value that acts as a flag to `rect_area()`. When this value is seen by the function, it uses the `length` parameter twice. Here is an example of this approach:

```

// Compute area of a rectangle using default arguments.
#include <iostream>
using namespace std;

// Return area of a rectangle.
double rect_area(double length, double width = 0)
{
    if(!width) width = length;
    return length * width;
}

int main()
{
    cout << "10 x 5.8 rectangle has area: ";
    cout << rect_area(10.0, 5.8) << '\n';

    cout << "10 x 10 square has area: ";
    cout << rect_area(10.0) << '\n';

    return 0;
}

```

Here 0 is the default value of `width`. This value was chosen because no rectangle will have a width of 0. (Actually, a rectangle with a width of 0 is a line.) Thus, if this default value

is seen, `rect_area()` automatically uses the value in `length` for the value of `width`.

As this example shows, default arguments often provide a simple alternative to function overloading. (Of course, there are many situations in which function overloading is still required.)

3. It is not only legal to give constructor functions default arguments, it is also common. As you saw earlier in this chapter, many times a constructor is overloaded simply to allow both initialized and uninitialized objects to be created. In many cases, you can avoid overloading a constructor by giving it one or more default arguments. For example, examine this program:

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

class myclass {
    int x;
public:
    /* Use default argument instead of overloading
       myclass's constructor. */
    myclass(int n = 0) { x = n; }
    int getx() { return x; }
};

int main()
{
    myclass o1(10); // declare with initial value
    myclass o2; // declare without initializer

    cout << "o1: " << o1.getx() << '\n';
    cout << "o2: " << o2.getx() << '\n';

    return 0;
}
```

As this example shows, by giving `n` the default value of 0, it is possible to create objects that have explicit initial values and those for which the default value is sufficient.

4. Another good application for a default argument is found when a parameter is used to select an option. It is possible to give that parameter a default value that is used as a flag that tells the function to continue to use the previously selected option. For example, in the following program, the function `print()`

C++

displays a string on the screen. If its **how** parameter is set to **ignore**, the text is displayed as is. If **how** is **upper**, the text is displayed in uppercase. If **how** is **lower**, the text is displayed in lowercase. When **how** is not specified, it defaults to -1, which tells the function to reuse the last **how** value.

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

const int ignore = 0;
const int upper = 1;
const int lower = 2;

void print(char *s, int how = -1);

int main()
{
    print("Hello There\n", ignore);
    print("Hello There\n", upper);
    print("Hello There\n"); // continue in upper
    print("Hello there\n", lower);
    print("That's all\n"); // continue in lower

    return 0;
}

/* Print a string in the specified case. Use
   last case specified if none is given.
*/
void print(char *s, int how)
{
    static int oldcase = ignore;

    // reuse old case if none specified
    if(how<0) how = oldcase;
    while(*s) {
        switch(how) {
            case upper: cout << (char) toupper(*s);
                       break;
            case lower: cout << (char) tolower(*s);
                       break;
            default: cout << *s;
        }
        s++;
    }
}
```

```
oldcase = how;
}
```

This function displays the following output:

```
Hello There
HELLO THERE
HELLO THERE
hello there
that's all
```

- Earlier in this chapter you saw the general form of a copy constructor. This general form was shown with only one parameter. However, it is possible to create copy constructors that take additional arguments, as long as the additional arguments have default values. For example, the following is also an acceptable form of a copy constructor:

```
myclass(const myclass &obj, int x=0) {
    // body of constructor
}
```

As long as the first argument is a reference to the object being copied, and all other arguments default, the function qualifies as a copy constructor. This flexibility allows you to create copy constructors that have other uses.

- Although default arguments are powerful and convenient, they can be misused. There is no question that, when used correctly, default arguments allow a function to perform its job in an efficient and easy-to-use manner. However, this is only the case when the default value given to a parameter makes sense. For example, if the argument is the value wanted nine times out ten, giving a function a default argument to this effect is obviously a good idea. However, in cases in which no one value is more likely to be used than another, or when there is no benefit to using a default argument as a flag value, it makes little sense to provide a default value. Actually, providing a default argument when one is not called for destructures your program and tends to mislead anyone else who has to use that function.

As with function overloading, part of becoming an excellent C++ programmer is knowing when to use a default argument and when not to.

EXERCISES

1. In the C++ standard library is the function `strtol()`, which has this prototype:

```
long strtol(const char *start, const **end, int base);
```

The function converts the numeric string pointed to by `start` into a long integer. The number base of the numeric string is specified by `base`. Upon return, `end` points to the character in the string immediately following the end of the number. The long integer equivalent of the numeric string is returned. `base` must be in the range 2 to 38. However, most commonly, base 10 is used.

Create a function called `mystrtol()` that works the same as `strtol()` except that `base` is given the default argument of 10. (Feel free to use `strtol()` to actually perform the conversion. It requires the header `<cstdlib>`.) Demonstrate that your version works correctly.

2. What is wrong with the following function prototype?

```
char *f(char *p, int x = 0, char *q);
```

3. Most C++ compilers supply nonstandard functions that allow cursor positioning and the like. If your compiler supplies such functions, create a function called `myclreol()` that clears the line from the current cursor position to the end of the line. However, give this function a parameter that specifies the number of character positions to clear. If the parameter is not specified, automatically clear the entire line. Otherwise, clear only the number of character positions specified by the parameter.
4. What is wrong with the following prototype, which uses a default argument?

```
int f(int count, int max = count);
```

5.5 OVERLOADING AND AMBIGUITY

When you are overloading functions, it is possible to introduce ambiguity into your program. Overloading-caused ambiguity can be introduced through type conversions, reference parameters, and default arguments. Further, some types of ambiguity are caused by the overloaded functions themselves. Other types occur in the manner in which an overloaded function is called. Ambiguity must be removed before your program will compile without error.

EXAMPLES

1. One of the most common types of ambiguity is caused by C++'s automatic type conversion rules. As you know, when a function is called with an argument that is of a compatible (but not the same) type as the parameter to which it is being passed, the type of the argument is automatically converted to the target type. In fact, it is this sort of type conversion that allows a function such as `putchar()` to be called with a character even though its argument is specified as an `int`. However, in some cases, this automatic type conversion will cause an ambiguous situation when a function is overloaded. To see how, examine this program:

```
// This program contains an ambiguity error.
#include <iostream>
using namespace std;

float f(float i)
{
    return i / 2.0;
}

double f(double i)
{
    return i / 3.0;
}

int main()
{
    float x = 10.09;
    double y = 10.09;
```

C++

```

cout << f(x); // unambiguous - use f(float)
cout << f(y); // unambiguous - use f(double)

cout << f(10); // ambiguous, convert 10 to double or float??

return 0;
}

```

As the comments in `main()` indicate, the compiler is able to select the correct version of `f()` when it is called with either a **float** or a **double** variable. However, what happens when it is called with an integer? Does the compiler call `f(float)` or `f(double)`? (Both are valid conversions!) In either case, it is valid to promote an integer into either a **float** or a **double**. Thus, the ambiguous situation is created.

This example also points out that ambiguity can be introduced by the way an overloaded function is called. The fact is that there is no inherent ambiguity in the overloaded versions of `f()` as long as each is called with an unambiguous argument.

- Here is another example of function overloading that is not ambiguous in and of itself. However, when this function is called with the wrong type of argument, C++'s automatic conversion rules cause an ambiguous situation.

```

// This program is ambiguous.
#include <iostream>
using namespace std;

void f(unsigned char c)
{
    cout << c;
}

void f(char c)
{
    cout << c;
}

int main()
{
    f('c');
    f(86); // which f() is called???
}

```

```
    return 0;
}
```

Here, when `f()` is called with the numeric constant 86, the compiler cannot know whether to call `f(unsigned char)` or `f(char)`. Either conversion is equally valid, thus leading to ambiguity.

3. One type of ambiguity is caused when you try to overload functions in which the only difference is the fact that one uses a reference parameter and the other uses the default call-by-value parameter. Given C++'s formal syntax, there is no way for the compiler to know which function to call. Remember, there is no syntactical difference between calling a function that takes a value parameter and calling a function that takes a reference parameter. For example:

```
// An ambiguous program.
#include <iostream>
using namespace std;

int f(int a, int b)
{
    return a+b;
}

// this is inherently ambiguous
int f(int a, int &b)
{
    return a-b;
}

int main()
{
    int x=1, y=2;

    cout << f(x, y); // which version of f() is called???

    return 0;
}
```

Here, `f(x, y)` is ambiguous because it could be calling either version of the function. In fact, the compiler will flag an error before this statement is even specified because the overloading

C++

of the two functions is inherently ambiguous and no reference to them could be resolved.

4. Another type of ambiguity is caused when you are overloading a function in which one or more overloaded functions use a default argument. Consider this program:

```
// Ambiguity based on default arguments plus overloading.
#include <iostream>
using namespace std;

int f(int a)
{
    return a*a;
}

int f(int a, int b = 0)
{
    return a*b;
}

int main()
{
    cout << f(10, 2); // calls f(int, int)
    cout << f(10); // ambiguous - call f(int) or f(int, int)???

    return 0;
}
```

Here the call **f(10, 2)** is perfectly acceptable and unambiguous. However, the compiler has no way of knowing whether the call **f(10)** is calling the first version of **f()** or the second version with **b** defaulting.

EXERCISE

1. Try to compile each of the preceding ambiguous programs. Make a mental note of the types of error messages they generate. This will help you recognize ambiguity errors when they creep into your own programs.
-

5.6

FINDING THE ADDRESS OF AN OVERLOADED FUNCTION

To conclude this chapter, you will learn how to find the address of an overloaded function. Just as in C, you can assign the address of a function (that is, its entry point) to a pointer and access that function via that pointer. A function's address is obtained by putting its name on the right side of an assignment statement without any parentheses or arguments. For example, if `zap()` is a function, assuming proper declarations, this is a valid way to assign `p` the address of `zap()`:

```
p = zap;
```

In C, any type of pointer can be used to point to a function because there is only one function that it can point to. However, in C++ the situation is a bit more complex because a function can be overloaded. Thus, there must be some mechanism that determines which function's address is obtained.

The solution is both elegant and effective. When obtaining the address of an overloaded function, it is *the way the pointer is declared* that determines which overloaded function's address will be obtained. In essence, the pointer's declaration is matched against those of the overloaded functions. The function whose declaration matches is the one whose address is used.

EXAMPLE

1. Here is a program that contains two versions of a function called `space()`. The first version outputs `count` number of spaces to the screen. The second version outputs `count` number of whatever type of character is passed to `ch`. In `main()`, two function pointers are declared. The first one is specified as a pointer to a function having only one integer parameter. The second is declared as a pointer to a function taking two parameters.

```
/* Illustrate assigning function pointers to
   overloaded functions. */
#include <iostream>
using namespace std;
```

▼ C++

```
// Output count number of spaces.
void space(int count)
{
    for( ; count; count--) cout << ' ';
}

// Output count number of chs.
void space(int count, char ch)
{
    for( ; count; count--) cout << ch;
}

int main()
{
    /* Create a pointer to void function with
       one int parameter. */
    void (*fp1)(int);

    /* Create a pointer to void function with
       one int parameter and one character parameter. */
    void (*fp2)(int, char);

    fp1 = space; // gets address of space(int)

    fp2 = space; // gets address of space(int, char)

    fp1(22); // output 22 spaces
    cout << "\n";

    fp2(30, 'x'); // output 30 x's
    cout << "\n";

    return 0;
}
```

As the comments illustrate, the compiler is able to determine which overloaded function to obtain the address of based upon how **fp1** and **fp2** are declared.

To review: When you assign the address of an overloaded function to a function pointer, it is the declaration of the pointer that determines which function's address is assigned. Further, the declaration of the function pointer must exactly match one and only one of the overloaded functions. If it does not, ambiguity will be introduced, causing a compile-time error.

EXERCISE

1. Following are two overloaded functions. Show how to obtain the address of each.

```
int dif(int a, int b)
{
    return a-b;
}
```

```
float dif(float a, float b)
{
    return a-b;
}
```

SKILLS CHECK

Mastery
Skills Check

At this point you should be able to perform the following exercises and answer the questions.

1. Overload the `date()` constructor from Section 5.1, Example 3, so that it accepts a parameter of type `time_t`. (Remember, `time_t` is a type defined by the standard time and date functions found in your C++ compiler's library.)
2. What is wrong with the following fragment?

```
class samp {
    int a;
public:
    samp(int i) { a = i; }
    // ...
}
```

```
int main()
{
    samp x, y(10);

    // ...
}
```

3. Give two reasons why you might want (or need) to overload a class's constructor.
4. What is the most common general form of a copy constructor?
5. What type of operations will cause the copy constructor to be invoked?
6. Briefly explain what the **overload** keyword does and why it is no longer needed.
7. Briefly describe a default argument.
8. Create a function called **reverse()** that takes two parameters. The first parameter, called **str**, is a pointer to a string that will be reversed upon return from the function. The second parameter is called **count**, and it specifies how many characters of **str** to reverse. Give **count** a default value that, when present, tells **reverse()** to reverse the entire string.
9. What is wrong with the following prototype?

```
char *wordwrap(char *str, int size=0, char ch);
```

10. Explain some ways that ambiguity can be introduced when you are overloading functions.
11. What is wrong with the following fragment?

```
void compute(double *num, int divisor=1);
void compute(double *num);
// ...
compute(&x);
```


12. When you are assigning the address of an overloaded function to a pointer, what is it that determines which version of the function is used?



This section checks how well you have integrated material in this chapter with that from the preceding chapters.

1. Create a function called **order()** that takes two integer reference parameters. If the first argument is greater than the second argument, reverse the two arguments. Otherwise, take no action. That is, order the two arguments used to call **order()** so that, upon return, the first argument will be less than the second. For example, given

```
int x=1, y=0;  
order(x, y);
```

following the call, **x** will be 0 and **y** will be 1.

2. Why are the following two overloaded functions inherently ambiguous?

```
int f(int a);  
int f(int &a);
```

3. Explain why using a default argument is related to function overloading.
4. Given the following partial class, add the necessary constructor functions so that both declarations within **main()** are valid. (Hint: You need to overload **samp()** twice.)

```
class samp {
    int a;
public:
    // add constructor functions
    int get_a() { return a; }
};

int main()
{
    samp ob(88); // init ob's a to 88
    samp obarray[10]; // noninitialized 10-element array

    // ...
}
```

5. Briefly explain why copy constructors are needed.