Part II UsingC++ Operators(二)

Using C++ Math Operators and Precedence

If you are dreading this chapter because you don't like math—relax, C++ does all your math for you! It is a misconception that you have to be good at math to understand how to program computers. In fact, programming practice assumes the opposite is true! Your computer is your "slave," to follow your instructions, and to do all the calculations for you. This chapter explains how C++ computes by introducing you to

- Primary math operators
- Order of operator precedence
- Assignment statements
- Mixed data type calculations
- Type casting

Many people who dislike math actually enjoy learning how the computer handles it. After learning the math operators and a few simple ways in which C++ uses them, you should feel comfortable using calculations in your programs. Computers are fast, and they can perform math operations much faster than you can!

C++'s Primary Math Operators

A C++ *math operator* is a symbol used for adding, subtracting, multiplying, dividing, and other operations. C++ operators are not always mathematical in nature, but many are. Table 8.1 lists these operator symbols and their primary meanings.

Table 8.1. C++ primary operators.

Symbol	*Meaning*
*	Multiplication
/	Division and Integer Division
%	Modulus or Remainder
+	Addition
-	Subtraction

Most of these operators work in the familiar way you expect them to. Multiplication, addition, and subtraction produce the same results (and the division operator *usually* does) as those produced with a calculator. Table 8.2 illustrates four of these simple operators.

Table 8.2. Typical operator results.

Formula	*Result*
`4 * 2`	8
`64 / 4`	16
`80 - 15`	65
`12 + 9`	21

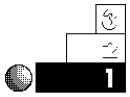
Table 8.2 contains examples of *binary operations* performed with the four operators. Don't confuse binary operations with *binary numbers.* When an operator is used between two literals, variables, or a combination of both, it is called a *binary operator* because it operates using two values. When you use these operators (when assigning their results to variables, for example), it does not matter in C++ whether you add spaces to the operators or not.

CAUTION: For multiplication, use the asterisk (`*`), *not* an `x` as you might normally do. An `x` cannot be used as the multiplication sign because C++ uses `x` as a variable name. C++ interprets `x` as the value of a variable called `x`.

The Unary Operators

A *unary operator* operates on, or affects, a single value. For instance, you can assign a variable a positive or negative number by using a unary + or –.

Examples



1. The following section of code assigns four variables a positive or a negative number. The plus and minus signs are all unary because they are not used between two values.

The variable `a` *is assigned a negative* `25` *value.*
The variable `b` *is assigned a positive* `25` *value.*
The variable `c` *is assigned a negative* `a` *value.*
The variable `d` *is assigned a positive* `b` *value.*

```
a = -25;// Assign 'a' a negative 25.
b = +25;// Assign 'b' a positive 25 (+ is not needed).
c = -a; // Assign 'c' the negative of 'a' (-25).
d = +b; // Assign 'd' the positive of 'b' (25, + not needed).
```

2. You generally do not have to use the unary plus sign. C++ assumes a number or variable is positive, even if it has no plus sign. The following four statements are equivalent to the previous four, except they do not contain plus signs.

```
a = -25;    // Assign 'a' a negative 25.
b =  25;    // Assign 'b' a positive 25.
c = -a;     // Assign 'c' the negative of 'a' (-25).
d =  b;     // Assign 'd' the positive of 'b' (25).
```

3. The unary negative comes in handy when you want to negate a single number or variable. The negative of a negative is positive. Therefore, the following short program assigns a negative number (using the unary –) to a variable, then prints the negative of that same variable. Because it had a negative number to begin with, the `cout` produces a positive result.

```
// Filename: C8NEG.CPP
// The negative of a variable that contains a negative value.
#include <iostream.h>
main()
{
   signed int temp=-12;  // 'signed' is not needed because
                         //     it is the default.
   cout << -temp << "\n";  // Produces a 12 on-screen.

   return 0;
}
```

The variable declaration does not need the *signed* prefix, because all integer variables are signed by default.

4. If you want to subtract the negative of a variable, make sure you put a space before the unary minus sign. For example, the following line:

```
new_temp + new_temp- -inversion_factor;
```

temporarily negates the `inversion_factor` and subtracts that negated value from `new_temp`.

Division and Modulus

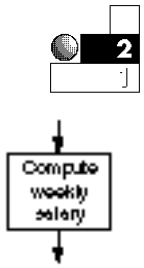
The division sign, /, and the modulus operator, %, might behave in ways unfamiliar to you. They're as easy to use, however, as the other operators you have just seen.

The modulus (%) computes remainders in division.

The forward slash (/) is always used for division. However, it produces an integer called `divide` if integer values (literals, variables, or a combination of both) appear on both sides of the slash. If there is a remainder, C++ discards it.

The percent sign (%) produces a *modulus*, or a *remainder*, of an integer division. It requires that integers be on both sides of the symbol, or it does not work.

Examples



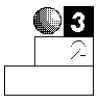
1. Suppose you want to compute your weekly pay. The following program asks for your yearly pay, divides it by 52, and prints the results to two decimal places.

```
// Filename: C8DIV.CPP
// Displays user's weekly pay.
#include <stdio.h>
main()
{
   float weekly, yearly;
   printf("What is your annual pay? ");  // Prompt user.
   scanf("%f", &yearly);

   weekly = yearly/52;  // Computes the weekly pay.
   printf("\n\nYour weekly pay is $%.2f", weekly);
   return 0;
}
```

Because a floating-point number is used in the division, C++ produces a floating-point result. Here is a sample output from such a program:

```
What is your annual pay? 38000.00
Your weekly pay is $730.77
```

Because this program used `scanf()` and `printf()` (to keep you familiar with both ways of performing input and output), the stdio.h header file is included rather than iostream.h.



2. Integer division does not round its results. If you divide two integers and the answer is not a whole number, C++ ignores the fractional part. The following `printf()`s help show this. The output that results from each `printf()` appears in the comment to the right of each line.

```
printf("%d \n", 10/2);      // 5  (no remainder)
printf("%d \n", 300/100);   // 3  (no remainder)
printf("%d \n", 10/3);      // 3  (discarded remainder)
printf("%d \n", 300/165);   // 1  (discarded remainder)
```

The Order of Precedence

Understanding the math operators is the first of two steps toward understanding C++ calculations. You must also understand the *order of precedence*. The order of precedence (sometimes called the *math hierarchy* or *order of operators*) determines exactly how C++ computes formulas. The precedence of operators is exactly the same concept you learned in high school algebra courses. (Don't worry, this is the easy part of algebra!) To see how the order of precedence works, try to determine the result of the following simple calculation:

```
2 + 3 * 2
```

If you said 10, you are not alone; many people respond with 10. However, 10 is correct only if you interpret the formula from the left. What if you calculated the multiplication first? If you took the value of `3 * 2` and got an answer of 6, then added the 2, you receive an answer of 8—which is exactly the same answer that C++ computes (and happens to be the correct way)!

C++ performs multiplication, division, and modulus *before* addition and subtraction.

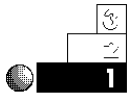
C++ always performs multiplication, division, and modulus first, then addition and subtraction. Table 8.3 shows the order of the operators you have seen so far. Of course, there are many more levels to C++'s precedence table of operators than the ones shown in Table 8.3. Unlike most computer languages, C++ has 20 levels of precedence. Appendix D, "C++ Precedence Table," contains the complete precedence table. Notice in this appendix that multiplication, division, and modulus reside on level 8, one level higher than

level 9's addition and subtraction. In the next few chapters, you learn how to use the remainder of this precedence table in your C++ programs.

Table 8.3. Order of precedence for primary operators.

Order	*Operator*
First	Multiplication, division, modulus remainder (*, /, %)
Second	Addition, subtraction (+, -)

Examples



1. It is easy to follow C++'s order of operators if you follow the intermediate results one at a time. The three calculations in Figure 8.1 show you how to do this.

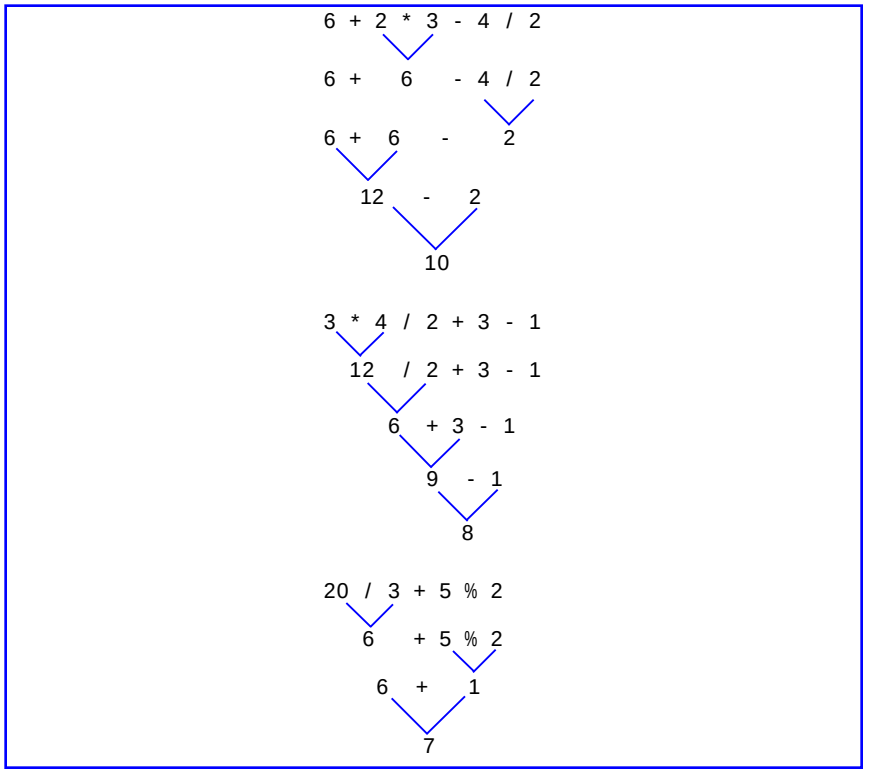


Figure 8.1. C++'s order of operators with lines indicating precedence.

2. Looking back at the order of precedence table, you might notice that multiplication, division, and modulus are on the same level. This implies there is no hierarchy on that level. If more than one of these operators appear in a calculation, C++ performs the math from the left. The same is true of addition and subtraction—C++ performs the operation on the extreme left first.

   Figure 8.2 illustrates an example showing this process.

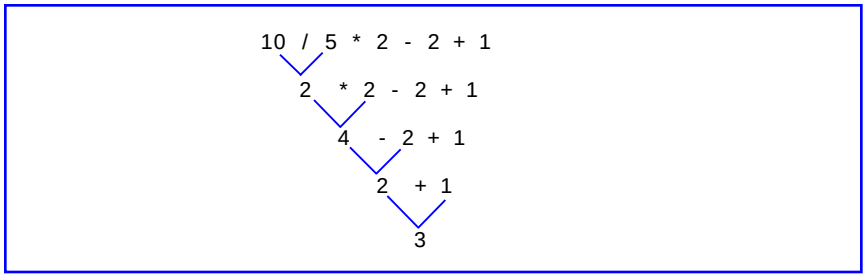


Figure 8.2. C++'s order of operators from the left, with lines indicating precedence.

Because the division appears to the left of the multiplication, it is computed first.

You now should be able to follow the order of these C++ operators. You don't have to worry about the math because C++ does the actual work. However, you should understand this order of operators so you know how to structure your calculations. Now that you have mastered this order, it's time to learn how you can override it with parentheses!

Using Parentheses

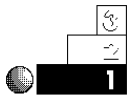
If you want to override the order of precedence, you can add parentheses to the calculation. The parentheses actually reside on a level above the multiplication, division, and modulus in the precedence table. In other words, any calculation in parentheses—whether it is addition, subtraction, division, or whatever—is always calculated before the rest of the line. The other calculations are then performed in their normal operator order.

Parentheses override the usual order of math.

The first formula in this chapter, `2 + 3 * 2`, produced an 8 because the multiplication was performed before addition. However, by adding parentheses around the addition, as in `(2 + 3) * 2`, the answer becomes 10.

In the precedence table shown in Appendix D, "C++ Precedence Table," the parentheses reside on level 3. Because they are higher than the other levels, the parentheses take precedence over multiplication, division, and all other operators.

Examples



1. The calculations shown in Figure 8.3 illustrate how parentheses override the regular order of operators. These are the same three formulas shown in the previous section, but their results are calculated differently because the parentheses override the normal order of operators.

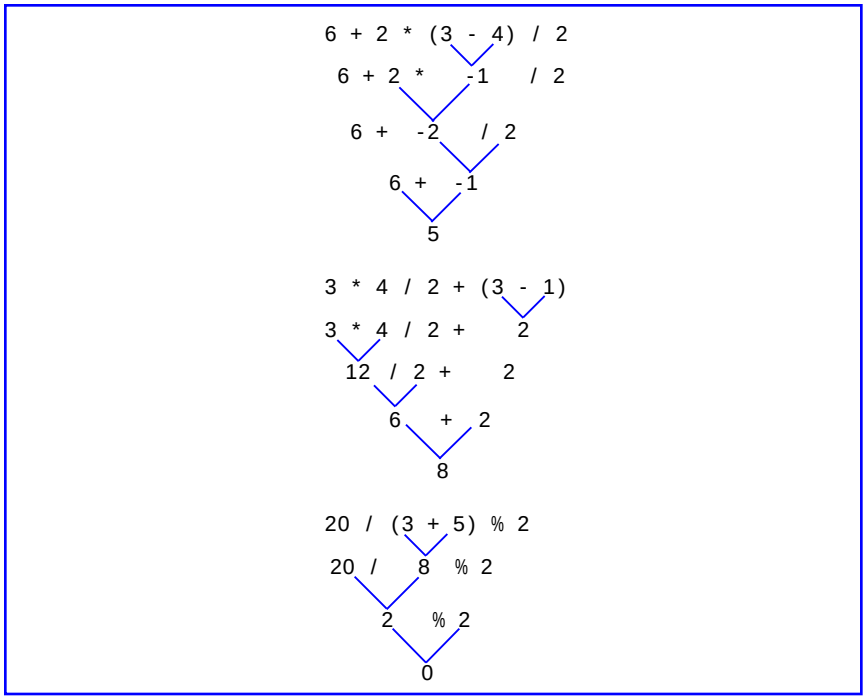


Figure 8.3. Example of parentheses as the highest precedence level with lines indicating precedence.

2. If an expression contains parentheses-within-parentheses, C++ evaluates the innermost parentheses first. The expressions in Figure 8.4 illustrate this.

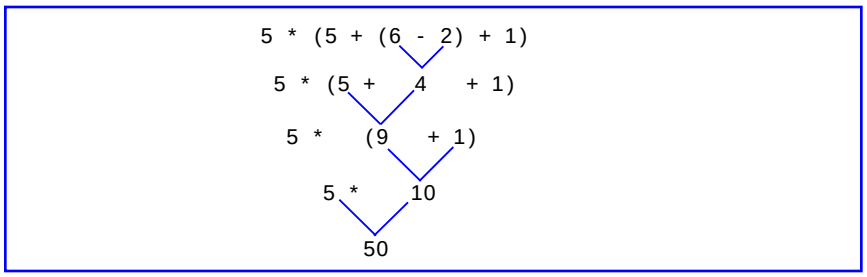


Figure 8.4. Precedence example of parentheses-within-parentheses with lines indicating precedence.

3. The following program produces an incorrect result, even though it looks as if it will work. See if you can spot the error!

Comments to identify your program.
Include the header file iostream.h so `cout` *works.*
Declare the variables `avg`, `grade1`, `grade2`, *and* `grade3` *as floating-point variables.*
The variable `avg` *becomes equal to* `grade3` *divided by 3.0 plus* `grade2` *plus* `grade1`.
Print to the screen `The average is` *and the average of the three grade variables.*
Return to the operating system.

```
// Filename: C8AVG1.CPP
// Compute the average of three grades.
#include <iostream.h>
main()
{
   float avg, grade1, grade2, grade3;

   grade1 = 87.5;
   grade2 = 92.4;
   grade3 = 79.6;
```

```
    avg = grade1 + grade2 + grade3 / 3.0;
    cout << "The average is " << avg << "\n";
    return 0;
}
```

The problem is that division is performed first. Therefore, the third grade is divided by 3.0 first, then the other two grades are added to that result. To correct this problem, you simply have to add one set of parentheses, as shown in the following:

```
// Filename: C8AVG2.CPP
// Compute the average of three grades.
#include <iostream.h>
main()
{
    float avg, grade1, grade2, grade3;

    grade1 = 87.5;
    grade2 = 92.4;
    grade3 = 79.6;

    avg = (grade1 + grade2 + grade3) / 3.0;
    cout << "The average is " << avg << "\n";
    return 0;
}
```

TIP: Use plenty of parentheses in your C++ programs to clarify the order of operators, even when you don't have to override their default order. Using parentheses makes the calculations easier to understand later, when you might have to modify the program.

Shorter Is Not Always Better

When you program computers for a living, it is much more important to write programs that are easy to understand than programs that are short or include tricky calculations.

Maintainability is the computer industry's word for the changing and updating of programs previously written in a simple style. The business world is changing rapidly, and the programs companies have used for years must often be updated to reflect this changing environment. Businesses do not always have the resources to write programs from scratch, so they usually modify the ones they have.

Years ago when computer hardware was much more expensive, and when computer memories were much smaller, it was important to write small programs, which often meant relying on clever, individualized tricks and shortcuts. Unfortunately, such programs are often difficult to revise, especially if the original programmers leave and someone else (you!) must modify the original code.

Companies are realizing the importance of spending time to write programs that are easy to modify and that do not rely on tricks, or "quick and dirty" routines that are hard to follow. You can be a much more valuable programmer if you write clean programs with ample white space, frequent remarks, and straightforward code. Use parentheses in formulas if it makes the formulas clearer, and use variables for storing results in case you need the same answer later in the program. Break long calculations into several smaller ones.

Throughout the remainder of this book, you can read tips on writing maintainable programs. You and your colleagues will appreciate these tips when you incorporate them in your own C++ programs.

The Assignment Statements

In C++, the assignment operator, =, behaves differently from what you might be used to in other languages. So far, you have used it to assign values to variables, which is consistent with its use in most other programming languages.

However, the assignment operator also can be used in other ways, such as multiple assignment statements and compound assignments, as the following sections illustrate.

Multiple Assignments

If two or more equal signs appear in an expression, each performs an assignment. This fact introduces a new aspect of the precedence order you should understand. Consider the following expression:

```
a=b=c=d=e=100;
```

This might at first seem confusing, especially if you know other computer languages. To C++, the equal sign always means: Assign the value on the right to the variable on the left. This right-to-left order is described in Appendix D's precedence table. The third column in the table is labeled *Associativity*, which describes the direction of the operation. The assignment operator associates from the right, whereas some of the other C++ operators associate from the left.

Because the assignment associates from the right, the previous expression assigns `100` to the variable named `e`. This assignment produces a value, `100`, for the expression. In C++, all expressions produce values, typically the result of assignments. Therefore, `100` is assigned to the variable `d`. The value, `100`, is assigned to `c`, then to `b`, and finally to `a`. The old values of these variables are replaced by `100` after the statement finishes.

Because C++ does not automatically set variables to zero before you use them, you might want to do so before you use the variables with a single assignment statement. The following section of variable declarations and initializations is performed using multiple assignment statements.

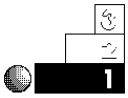
```
main()
{
   int ctr, num_emp, num_dep;
   float sales, salary, amount;

   ctr=num_emp=num_dep=0;
   sales=salary=amount=0;
   // Rest of program follows.
```

In C++, you can include the assignment statement almost anywhere in a program, even in another calculation. For example, consider this statement:

```
value = 5 + (r = 9 - c);
```

which is a perfectly legal C++ statement. The assignment operator resides on the first level of the precedence table, and always produces a value. Because its associativity is from the right, the `r` is assigned `9 - c` because the equal sign on the extreme right is evaluated first. The subexpression (`r = 9 - c`) produces a value (and places that value in `r`), which is then added to `5` before storing the answer in `value`.

Example



Because C++ does not initialize variables to zero before you use them, you might want to include a multiple assignment operator to do so before using the variables. The following section of code ensures that all variables are initialized before the rest of the program uses them.

```
main()
{
   int num_emp, dependents, age;
   float salary, hr_rate, taxrate;

   // Initialize all variables to zero.
   num_emp=dependents=age=hours=0;
   salary=hr_rate=taxrate=0.0;

   // Rest of program follows.
```

Compound Assignments

Many times in programming, you might want to update the value of a variable. In other words, you have to take a variable's current value, add or multiply that value by an expression, then reassign it to the original variable. The following assignment statement demonstrates this process:

```
salary=salary*1.2;
```

This expression multiplies the old value of `salary` by 1.2 (in effect, raising the value in `salary` by 20 percent), then reassigns it to `salary`. C++ provides several operators, called *compound operators*, that you can use any time the same variable appears on both sides of the equal sign. The compound operators are shown in Table 8.4.

Table 8.4. C++'s compound operators.

Operator	*Example*	*Equivalent*
`+=`	`bonus+=500;`	`bonus=bonus+500;`
`-=`	`budget-=50;`	`budget=budget-50;`
`*=`	`salary*=1.2;`	`salary=salary*1.2;`
`/=`	`factor/=.50;`	`factor=factor/.50;`
`%=`	`daynum%=7;`	`daynum=daynum%7;`

The compound operators are low in the precedence table. They typically are evaluated last or near-last.

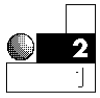
Examples

1. You have been storing your factory's production amount in a variable called `prod_amt`, and your supervisor has just informed you that a new addition has to be applied to the production value. You could code this update in a statement, as follows:

   ```
   prod_amt = prod_amt + 2.6;  // Add 2.6 to current production.
   ```

   Instead of using this formula, use C++'s compound addition operator by coding it like this:

   ```
   prod_amt += 2.6;  // Add 2.6 to current production.
   ```



2. Suppose you are a high school teacher who wants to raise your students' grades. You gave a test that was too difficult, and the grades were not what you expected. If you had stored each of the student's grades in variables named `grade1`, `grade2`, `grade3`, and so on, you can update the grades in a program with the following section of compound assignments.

```
grade1*=1.1;      // Increase each student's grade by 10.
percent.
grade2*=1.1;
grade3*=1.1;
// Rest of grade changes follow.
```

3. The precedence of the compound operators requires important consideration when you decide how to code compound assignments. Notice from Appendix D, "C++ Precedence Table," that the compound operators are on level 19, much lower than the regular math operators. This means you must be careful how you interpret them.

   For example, suppose you want to update the value of a `sales` variable with this formula:

   ```
   4-factor+bonus
   ```

   You can update the `sales` variable with the following statement:

   ```
   sales = *4 - factor + bonus;
   ```

   This statement adds the quantity `4-factor+bonus` to `sales`. Due to operator precedence, this statement is not the same as the following one:

   ```
   sales = sales *4 - factor + bonus;
   ```

   Because the `*=` operator is much lower in the precedence table than `*` or `-`, it is performed last, and with right-to-left associativity. Therefore, the following are equivalent, from a precedence viewpoint:

   ```
   sales *= 4 - factor + bonus;
   ```

   and

   ```
   sales = sales * (4 - factor + bonus);
   ```

Mixing Data Types in Calculations

You can mix data types in C++. Adding an integer and a floating-point value is mixing data types. C++ generally converts

the smaller of the two types into the other. For instance, if you add a double to an integer, C++ first converts the integer into a double value, then performs the calculation. This method produces the most accurate result possible. The automatic conversion of data types is only temporary; the converted value is back in its original data type as soon as the expression is finished.

C++ attempts to convert the smaller data type to the larger one in a mixed data-type expression.

If C++ converted two different data types to the smaller value's type, the higher-precision value is *truncated*, or shortened, and accuracy is lost. For example, in the following short program, the floating-point value of `sales` is added to an integer called `bonus`. Before C++ computes the answer, it converts `bonus` to floating-point, which results in a floating-point answer.

```
// Filename: C8DATA.CPP
// Demonstrate mixed data type in an expression.
#include <stdio.h>
main()
{
   int bonus=50;
   float salary=1400.50;
   float total;

   total=salary+bonus;  // bonus becomes floating-point
                        // but only temporarily.
   printf("The total is %.2f", total);
   return 0;
}
```

Type Casting

Most of the time, you don't have to worry about C++'s automatic conversion of data types. However, problems can occur if you mix unsigned variables with variables of other data types. Due to differences in computer architecture, unsigned variables do not always convert to the larger data type. This can result in loss of accuracy, and even incorrect results.

You can override C++'s default conversions by specifying your own temporary type change. This process is called *type casting*. When you type cast, you temporarily change a variable's data type

from its declared data type to a new one. There are two formats of the type cast. They are

```
(data type) expression
```

and

```
data type(expression)
```

where `data type` can be any valid C++ data type, such as `int` or `float`, and the `expression` can be a variable, literal, or an expression that combines both. The following code temporarily type casts the integer variable `age` into a double floating-point variable, so it can be multiplied by the double floating-point `factor`. Both formats of the type cast are illustrated.

The variable `age_factor` *is assigned the value of the variable* `age` *(now treated like a double floating-point variable) multiplied by the variable* `factor`.

```
age_factor = (double)age * factor;    // Temporarily change age
                                      // to double.
```

The second way of type casting adds the parentheses around the variable rather than the data type, as so:

```
age_factor = double(age) * factor;    // Temporarily change age
                                      // to double.
```

NOTE: Type casting by adding the parentheses around the expression and not the data type is new to C++. C programmers do not have the option—they must put the data type in parentheses. The second method "feels" like a function call and seems to be more natural for this language. Therefore, becoming familiar with the second method will clarify your code.

Examples

1. Suppose you want to verify the interest calculation used by your bank on a loan. The interest rate is 15.5 percent, stored as .155 in a floating-point variable. The amount of interest you owe is computed by multiplying the interest rate by the amount of the loan balance, then multiplying that by the number of days in the year since the loan originated. The following program finds the daily interest rate by dividing the annual interest rate by 365, the number of days in a year. C++ must convert the integer 365 to a floating-point literal automatically, because it is used in combination with a floating-point variable.

```
// Filename: C8INT1.CPP
// Calculate interest on a loan.
#include <stdio.h>
main()
{
   int days=45;  // Days since loan origination.
   float principle = 3500.00; // Original loan amount
   float interest_rate=0.155;   // Annual interest rate
   float daily_interest;   // Daily interest rate

   daily_interest=interest_rate/365; // Compute floating-
                                     // point value.

   // Because days is int, it too is converted to float.
   daily_interest = principle * daily_interest * days;
   principle+=daily_interest;//Update principle with interest.
   printf("The balance you owe is %.2f\n", principle);
   return 0;
}
```

The output of this program follows:

```
The balance you owe is 3566.88
```

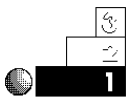
2. Instead of having C++ perform the conversion, you might want to type cast all mixed expressions to ensure they convert to your liking. Here is the same program as in the first example, except type casts are used to convert the integer literals to floating-points before they are used.

```
// Filename: C8INT2.CPP
// Calculate interest on a loan using type casting.
#include <stdio.h>
main()
{
   int days=45;  // Days since loan origination.
   float principle = 3500.00;   // Original loan amount
   float interest_rate=0.155;   // Annual interest rate
   float daily_interest;        // Daily interest rate

   daily_interest=interest_rate/float(365);  // Type cast days
                                             // to float.

   // Because days is integer, convert it to float also.
   daily_interest = principle * daily_interest * float(days);
   principle+=daily_interest;// Update principle with interest.
   printf("The balance you owe is %.2f", principle);
   return 0;
}
```

The output from this program is exactly the same as the previous one.

Review Questions

The answers to the review questions are in Appendix B.



1. What is the result for each of the following expressions?

   a. `1 + 2 * 4 / 2`

   b. `(1 + 2) * 4 / 2`

   c. `1 + 2 * (4 / 2)`

2. What is the result for each of the following expressions?

   a. `9 % 2 + 1`

   b. `(1 + (10 - (2 + 2)))`

3. Convert each of the following formulas into its C++ assignment equivalents.

   a. $a = \frac{3 + 3}{4 + 4}$

   b. `x = (a - b)*(a - c)2`

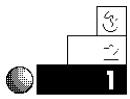
   c. $f = \frac{a2}{b3}$

   d. $d = \frac{(8 - x2)}{(x - 9)} - \frac{(4 * 2 - 1)}{x3}$

4. Write a short program that prints the area of a circle, when its radius equals 4 and π equals 3.14159. (*Hint:* The area of a circle is computed by π * radius^2.)

5. Write the assignment and `printf()` statements that print the remainder of 100/3.

Review Exercises



1. Write a program that prints each of the first eight powers of 2 (21, 22, 23,...28). Please write comments and include your name at the top of the program. Print string literals that describe each answer printed. The first two lines of your output should look like this:

```
2 raised to the first power is 2
2 raised to the second power is 4
```

2. Change C8PAY.CPP so it computes and prints a bonus of 15 percent of the gross pay. Taxes are not to be taken out of the bonus. After printing the four variables, `gross_pay`, `tax_rate`, `bonus`, and `gross_pay`, print a check on-screen that looks like a printed check. Add string literals so it prints the check-holder and put your name as the payer at the bottom of the check.

3. Store the weights and ages of three people in variables. Print a table, with titles, of the weights and ages. At the bottom of the table, print the averages.

4. Assume that a video store employee works 50 hours. He is paid $4.50 for the first 40 hours, time-and-a-half (1.5 times the regular pay rate) for the first five hours over 40, and double-time pay for all hours over 45. Assuming a 28 percent tax rate, write a program that prints his gross pay, taxes, and net pay to the screen. Label each amount with appropriate titles (using string literals) and add appropriate comments in the program.

Summary

You now understand C++'s primary math operators and the importance of the precedence table. Parentheses group operations so they can override the default precedence levels. Unlike some other programming languages, every operator in C++ has a meaning, no matter where it appears in an expression. This fact enables you to use the assignment operator (the equal sign) in the middle of other expressions.

When you perform math with C++, you also must be aware of how C++ interprets data types, especially when you mix them in the same expression. Of course, you can temporarily type cast a variable or literal so you can override its default data type.

This chapter has introduced you to a part of the book concerned with C++ operators. The following two chapters (Chapter 9, "Relational Operators," and Chapter 10, "Logical Operators") extend this introduction to include relational and logical operators. They enable you to compare data and compute accordingly.

Relational Operators

Sometimes you won't want every statement in your C++ program to execute every time the program runs. So far, every program in this book has executed from the top and has continued, line-by-line, until the last statement completes. Depending on your application, you might not always want this to happen.

Programs that don't always execute by rote are known as *data-driven* programs. In data-driven programs, the data dictates what the program does. You would not want the computer to print every employee's paychecks for every pay period, for example, because some employees might be on vacation, or they might be paid on commission and not have made a sale during that period. Printing paychecks with zero dollars is ridiculous. You want the computer to print checks only for employees who have worked.

This chapter shows you how to create data-driven programs. These programs do not execute the same way every time. This is possible through the use of *relational* operators that *conditionally* control other statements. Relational operators first "look" at the literals and variables in the program, then operate according to what they "find." This might sound like difficult programming, but it is actually straightforward and intuitive.

This chapter introduces you to

- Relational operators
- The `if` statement
- The `else` statement

Not only does this chapter introduce these comparison commands, but it prepares you for much more powerful programs, possible once you learn the relational operators.

Defining Relational Operators

Relational operators compare data.

In addition to the math operators you learned in Chapter 8, "Using C++ Math Operators and Precedence," there are also operators that you use for data comparisons. They are called *relational operators*, and their task is to compare data. They enable you to determine whether two variables are equal, not equal, and which one is less than the other. Table 9.1 lists each relational operator and its meaning.

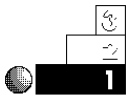
Table 9.1. The relational operators.

Operator	*Description*
==	Equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to
!=	Not equal to

The six relational operators form the foundation of data comparison in C++ programming. They always appear with two literals, variables, expressions (or some combination of these), one on each side of the operator. These relational operators are useful and you should know them as well as you know the `+`, `-`, `*`, `/`, and `%` mathematical operators.

NOTE: Unlike many programming languages, C++ uses a double equal sign (==) as a test for equality. The single equal sign (=) is reserved for assignment of values.

Examples



1. Assume that a program initializes four variables as follows:

```
int a=5;
int b=10;
int c=15;
int d=5;
```

The following statements are then True:

`a` is equal to `d`, so `a == d`

`b` is less than `c`, so `b < c`

`c` is greater than `a`, so `c > a`

`b` is greater than or equal to `a`, so `b >= a`

`d` is less than or equal to `b`, so `d <= b`

`b` is not equal to `c`, so `b != c`

These are not C++ statements; they are statements of comparison (*relational logic*) between values in the variables. Relational logic is easy.

Relational logic always produces a *True* or *False* result. In C++, unlike some other programming languages, you can directly use the True or False result of relational operators inside other expressions. You will soon learn how to do this; but for now, you have to understand only that the following True and False evaluations are correct:

- A True relational result evaluates to 1.
- A False relational result evaluates to 0.

Each of the statements presented earlier in this example evaluates to a 1, or True, result.

2. If you assume the same values as stated for the previous example's four variables, each of the value's statements is False (0):

```
a == b
b > c

d < a

d > a

a != d

b >= c

c <= b
```

Study these statements to see why each is False and evaluates to 0. The variables `a` and `d`, for example, are exactly equal to the same value (5), so neither is greater or less than the other.

You use relational logic in everyday life. Think of the following statements:

"The generic butter costs less than the name brand."

"My child is younger than Johnny."

"Our salaries are equal."

"The dogs are not the same age."

Each of these statements can be either True or False. There is no other possible answer.

Watch the Signs!

Many people say they are "not math-inclined" or "not logical," and you might be one of them. But, as mentioned in Chapter 8, you do not have to be good in math to be a good computer programmer. Neither should you be frightened by the term

"relational logic," because you just saw how you use it in everyday life. Nevertheless, symbols confuse some people.

The two primary relational operators, *less than* (<) and *greater than* (>), are easy to remember. You probably learned this concept in school, but might have forgotten it. Actually, their signs tell you what they mean.

The arrow points to the lesser of the two values. Notice how, in the previous Example 1, the arrow (the point of the < or >) always points to the lesser number. The larger, open part of the arrow points to the larger number.

The relation is False if the arrow is pointing the wrong way. In other words, `4 > 9` is False because the operator symbol is pointing to the 9, which is not the lesser number. In English this statement says, "4 is greater than 9," which is clearly false.

The if Statement

You incorporate relational operators in C++ programs with the if statement. Such an expression is called a *decision statement* because it tests a relationship—using the relational operators—and, based on the test's result, makes a decision about which statement to execute next.

The `if` statement appears as follows:

```
if (condition)
  { block of one or more C++ statements }
```

The `condition` includes any relational comparison, and it must be enclosed in parentheses. You saw several relational comparisons earlier, such as `a==d`, `c<d`, and so on. The `block of one or more C++ statements` is any C++ statement, such as an assignment or `printf()`, enclosed in braces. The block of the `if`, sometimes called the *body* of the `if` statement, is usually indented a few spaces for readability. This enables you to see, at a glance, exactly what executes if `condition` is True.

If only one statement follows the `if`, the braces are not required (but it is always good to include them). The block executes only if `condition` is True. If `condition` is False, C++ ignores the block and simply executes the next appropriate statement in the program that follows the `if` statement.

The `if` statement makes a decision.

Basically, you can read an `if` statement in the following way: "If the condition is True, perform the block of statements inside the braces. Otherwise, the condition must be False; so do not execute that block, but continue executing the remainder of the program as though this `if` statement did not exist."

The `if` statement is used to make a decision. The block of statements following the `if` executes if the decision (the result of the relation) is True, but the block does not execute otherwise. As with relational logic, you also use if logic in everyday life. Consider the statements that follow.

"If the day is warm, I will go swimming."

"If I make enough money, we will build a new house."

"If the light is green, go."

"If the light is red, stop."

Each of these statements is *conditional.* That is, if *and only if* the condition is true do you perform the activity.

CAUTION: Do not type a semicolon after the parentheses of the relational test. Semicolons appear after each statement inside the block.

Expressions as the Condition

C++ interprets any nonzero value as True, and zero always as False. This enables you to insert regular nonconditional expressions in the `if` logic. To understand this concept, consider the following section of code:

```
main()

{

   int age=21;   // Declares and assigns age as 21.

   if (age=85)

   {  cout << "You have lived through a lot!"; }

   // Remaining program code goes here.
```

At first, it might seem as though the `printf()` does not execute, but it does! Because the code line used a regular assignment operator (=) (not a relational operator, ==), C++ performs the assignment of `85` to `age`. This, as with all assignments you saw in Chapter 8, "Using C++ Math Operators and Precedence," produces a value for the expression of `85`. Because `85` is nonzero, C++ interprets the `if` condition as True and then performs the body of the `if` statement.

Confusing the relational equality test (==) with the regular assignment operator (=) is a common error in C++ programs, and the nonzero True test makes this bug even more difficult to find.

The designers of C++ didn't intend for this to confuse you. They want you to take advantage of this feature whenever you can. Instead of putting an assignment before an `if` and testing the result of that assignment, you can combine the assignment and `if` into a single statement.

Test your understanding of this by considering this: Would C++ interpret the following condition as True or False?

```
if (10 == 10 == 10)...
```

Be careful! At first glance, it seems True; but C++ interprets it as False! Because the == operator associates from the left, the program compares the first `10` to the second. Because they are equal, the result is `1` (for True) and the `1` is then compared to the third `10`—which results in a `0` (for False)!

Examples

1. The following are examples of valid C++ `if` statements.

If (the variable `sales` *is greater than* `5000`*), then the variable* `bonus` *becomes equal to* `500`.

```
if (sales > 5000)
   { bonus = 500; }
```

If this is part of a C++ program, the value inside the variable `sales` determines what happens next. If `sales` contains more than `5000`, the next statement that executes is the one inside the block that initializes `bonus`. If, however, `sales` contains `5000` or less, the block does not execute, and the line following the `if`'s block executes.

If (the variable `age` *is less than or equal to* `21`*) then print* `You are a minor.` *to the screen and go to a new line, print* `What is your grade?` *to the screen, and accept an integer from the keyboard.*

```
if (age <= 21)
   { cout << "You are a minor.\n";
     cout << "What is your grade? ";
     cin >> grade; }
```

If the value in `age` is less than or equal to `21`, the lines of code within the block execute next. Otherwise, C++ skips the entire block and continues with the remaining program.

If (the variable `balance` *is greater than the variable* `low_balance`*), then print* `Past due!` *to the screen and move the cursor to a new line.*

```
if (balance > low_balance)
   {cout << "Past due!\n"; }
```

If the value in `balance` is more than that in `low_balance`, execution of the program continues at the block and the message "`Past due!`" prints on-screen. You can compare two variables to each other (as in this example), or a variable to a literal (as in the previous examples), or a literal to a literal (although this is rarely done), or a literal to any expression in place of any variable or literal. The following `if` statement shows an expression included in the `if`.

If (the variable pay *multiplied by the variable* tax_rate *equals the variable* minimum*), then the variable* low_salary *is assigned* 1400.60.

```
If (pay * tax_rate == minimum)
   { low_salary = 1400.60; }
```

The precedence table of operators in Appendix D, "C++ Precedence Table," includes the relational operators. They are at levels 11 and 12, lower than the other primary math operators. When you use expressions such as the one shown in this example, you can make these expressions much more readable by enclosing them in parentheses (even though C++ does not require it). Here is a rewrite of the previous `if` statement with ample parentheses:

If (the variable pay *(multiplied by the variable* tax_rate*) equals the variable* minimum*), then the variable* low_salary *is assigned* 1400.60.

```
If ((pay * tax_rate) == minimum)
   { low_salary = 1400.60; }
```

2. The following is a simple program that computes a salesperson's pay. The salesperson receives a flat rate of $4.10 per hour. In addition, if sales are more than $8,500, the salesperson also receives an additional $500 as a bonus. This is an introductory example of conditional logic, which depends on a relation between two values, sales and $8500.

```
// Filename: C9PAY1.CPP
// Calculates a salesperson's pay based on his or her sales.
#include <iostream.h>
#include <stdio.h>
main()
{
   char sal_name[20];
   int hours;
   float total_sales, bonus, pay;

   cout << "\n\n";        // Print two blank lines.
   cout << "Payroll Calculation\n";
   cout << "-------------------\n";
```

```
   // Ask the user for needed values.
   cout << "What is salesperson's last name? ";
   cin >> sal_name;
   cout << "How many hours did the salesperson work? ";
   cin >> hours;
   cout << "What were the total sales? ";
   cin >> total_sales;

   bonus = 0;     // Initially, there is no bonus.

   // Compute the base pay.
   pay = 4.10 * (float)hours;  // Type casts the hours.

   // Add bonus only if sales were high.
   if (total_sales > 8500.00)
      { bonus = 500.00; }

   printf("%s made $%.2f \n", sal_name, pay);
   printf("and got a bonus of $%.2f", bonus);

   return 0;
}
```

This program uses `cout`, `cin`, and `printf()` for its input and output. You can mix them. Include the appropriate header files if you do (stdio.h and iostream.h).

The following output shows the result of running this program twice, each time with different input values. Notice that the program does two different things: It computes a bonus for one employee, but doesn't for the other. The $500 bonus is a direct result of the `if` statement. The assignment of `$500` to `bonus` executes only if the value in `total_sales` is more than `$8500`.

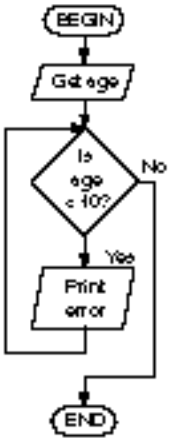
```
Payroll Calculation
-------------------
What is salesperson's last name? Harrison
How many hours did the salesperson work? 40
What were the total sales? 6050.64
Harrison made $164.00
and got a bonus of $0.00
```

```
Payroll Calculation
-------------------
What is salesperson's last name? Robertson
How many hours did the salesperson work? 40
What were the total sales? 9800
Robertson made $164.00
and got a bonus of $500.00
```

3. When programming the way users input data, it is wise to program *data validation* on the values they type. If they enter a bad value (for instance, a negative number when the input cannot be negative), you can inform them of the problem and ask them to reenter the data.

   Not all data can be validated, of course, but most of it can be checked for reasonableness. For example, if you write a student record-keeping program, to track each student's name, address, age, and other pertinent data, you can check whether the age falls in a reasonable range. If the user enters `213` for the age, you know the value is incorrect. If the user enters `-4` for the age, you know this value is also incorrect. Not all erroneous input for age can be checked, however. If the user is 21, for instance, and types `22`, your program has no way of knowing whether this is correct, because 22 falls in a reasonable age range for students.

   The following program is a routine that requests an age, and makes sure it is more than 10. This is certainly not a fool-proof test (because the user can still enter incorrect ages), but it takes care of extremely low values. If the user enters a bad age, the program asks for it again inside the `if` statement.




```
// Filename: C9AGE.CPP
// Program that ensures age values are reasonable.
#include <stdio.h>
main()
{
   int age;

   printf("\nWhat is the student's age? ");
   scanf(" %d", &age);  // With scanf(), remember the &
```

```
    if (age < 10)
      { printf("%c", '\x07');   // BEEP
        printf("*** The age cannot be less than 10 ***\n");
        printf("Try again...\n\n");
        printf("What is the student's age? ");
        scanf(" %d", &age);
      }

    printf("Thank you. You entered a valid age.");
    return 0;
}
```

This routine can also be a section of a longer program. You learn later how to prompt repeatedly for a value until a valid input is given. This program takes advantage of the bell (ASCII 7) to warn the user that a bad age was entered. Because the `\a` character is an escape sequence for the alarm (see Chapter 4, "Variables and Literals" for more information on escape sequences), `\a` can replace the `\x07` in this program.

If the entered `age` is less than `10`, the user receives an error message. The program beeps and warns the user about the bad `age` before asking for it again.

The following shows the result of running this program. Notice that the program "knows," due to the `if` statement, whether `age` is more than `10`.

```
What is the student's age? 3
*** The age cannot be less than 10 ***
Try again...

What is the student's age? 21
Thank you. You entered a valid age.
```

4. Unlike many languages, C++ does not include a square math operator. Remember that you "square" a number by multiplying it times itself (`3*3`, for example). Because many computers do not allow for integers to hold more than the square of 180, the following program uses `if` statements to make sure the number fits as an integer.

EXAMPLE

The program takes a value from the user and prints its square—unless it is more than 180. The message `* Square is not allowed for numbers over 180 *` appears on-screen if the user types a huge number.

```
// Filename: C9SQR1.CPP
// Print the square of the input value
// if the input value is less than 180.
#include <iostream.h>
main()
{
   int num, square;

   cout << "\n\n";  // Print two blank lines.
   cout << "What number do you want to see the square of? ";
   cin >> num;

   if (num <= 180)
   { square = num * num;
     cout << "The square of " << num << " is " <<
             square << "\n";
   }

   if (num > 180)
   { cout << '\x07';   // BEEP
  cout << "\n* Square is not allowed for numbers over 180 *";
  cout << "\nRun this program again trying a smaller value.";
   }

   cout << "\nThank you for requesting square roots.\n";
   return 0;
}
```

The following output shows a couple of sample runs with this program. Notice that both conditions work: If the user enters a number less than 180, the calculated square appears, but if the user enters a larger number, an error message appears.

```
What number do you want to see the square of? 45

The square of 45 is 2025
Thank you for requesting square roots.

What number do you want to see the square of? 212

* Square is not allowed for numbers over 180 *
Run this program again trying a smaller value.
Thank you for requesting square roots.
```

You can improve this program with the `else` statement, which you learn later in this chapter. This code includes a redundant check of the user's input. The variable `num` must be checked once to print the square if the input number is less than or equal to 180, and checked again for the error message if it is greater than 180.

5. The value of 1 and 0 for True and False, respectively, can help save you an extra programming step, which you are not necessarily able to save in other languages. To understand this, examine the following section of code:

```
commission = 0;    // Initialize commission

if (sales > 10000)
   { commission = 500.00; }

pay = net_pay + commission;    // Commission is 0 unless
                               // high sales.
```

You can make this program more efficient by combining the `if`'s relational test because you know that `if` returns 1 or 0:

```
pay = net_pay + (commission = (sales > 10000) * 500.00);
```

This single line does what it took the previous four lines to do. Because the assignment on the extreme right has precedence, it is computed first. The program compares the variable `sales` to `10000`. If it is more than `10000`, a True result of 1 returns. The program then multiplies 1 by `500.00` and stores the result in `commission`. If, however, the `sales` were not

more than 10000, a 0 results and the program receives 0 from multiplying 0 by 500.00.

Whichever value (500.00 or 0) the program assigns to commission is then added to net_pay and stored in pay.

The else Statement

The else statement never appears in a program without an if statement. This section introduces the else statement by showing you the popular if-else combination statement. Its format is

```
if (condition)
  { A block of 1 or more C++ statements }
else
  { A block of 1 or more C++ statements }
```

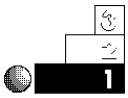
The first part of the if-else is identical to the if statement. If condition is True, the block of C++ statements following the if executes. However, if condition is False, the block of C++ statements following the else executes instead. Whereas the simple if statement determines what happens only when the condition is True, the if-else also determines what happens if the condition is False. No matter what the outcome is, the statement following the if-else executes next.

The following describes the nature of the if-else:

- If the condition test is True, the entire block of statements following the if executes.
- If the condition test is False, the entire block of statements following the else executes.

NOTE: You can also compare characters, in addition to numbers. When you compare characters, C++ uses the ASCII table to determine which character is “less than” the other (lower in the ASCII table). But you cannot compare character strings or arrays of character strings directly with relational operators.

Examples



1. The following program asks the user for a number. It then prints whether or not the number is greater than zero, using the `if-else` statement.

```
// Filename: C9IFEL1.CPP
// Demonstrates if-else by printing whether an
// input value is greater than zero or not.
#include <iostream.h>
main()
{
   int num;

   cout << "What is your number? ";
   cin >> num;   // Get the user's number.

   if (num > 0)
      { cout << "More than 0\n"; }
   else
      { cout << "Less or equal to 0\n"; }

   // No matter what the number was, the following executes.
   cout << "\n\nThanks for your time!\n";
   return 0;
}
```

There is no need to test for both possibilities when you use an `else`. The `if` tests whether the number is greater than zero, and the `else` automatically handles all other possibilities.

2. The following program asks the user for his or her first name, then stores it in a character array. The program checks the first character of the array to see whether it falls in the first half of the alphabet. If it does, an appropriate message is displayed.

```
// Filename: C9IFEL2.CPP
// Tests the user's first initial and prints a message.
#include <iostream.h>
main()
{
```

```
   char last[20];   // Holds the last name.
   cout << "What is your last name? ";
   cin >> last;

   // Test the initial
   if (last[0] <= 'P')
      { cout << "Your name is early in the alphabet.\n";}
   else
      { cout << "You have to wait a while for "
             << "YOUR name to be called!\n";}
   return 0;
}
```

Notice that because the program is comparing a character array element to a character literal, you must enclose the character literal inside single quotation marks. The data type on each side of each relational operator must match.

3. The following program is a more complete payroll routine than the other one. It uses the `if` statement to illustrate how to compute overtime pay. The logic goes something like this:

   If employees work 40 hours or fewer, they are paid regular pay (their hourly rate times the number of hours worked). If employees work between 40 and 50 hours, they receive one-and-a-half times their hourly rate for those hours over 40, in addition to their regular pay for the first 40. All hours over 50 are paid at double the regular rate.

```
// Filename: C9PAY2.CPP
// Compute the full overtime pay possibilities.
#include <iostream.h>
#include <stdio.h>
main()
{
   int hours;
   float dt, ht, rp, rate, pay;

   cout << "\n\nHow many hours were worked? ";
   cin >> hours;
   cout << "\nWhat is the regular hourly pay? ";
   cin >> rate;
```

```
   // Compute pay here
   // Double-time possibility
   if (hours > 50)
     { dt = 2.0 * rate * (float)(hours - 50);
       ht = 1.5 * rate * 10.0;}  // Time + 1/2 for 10 hours.
   else
     { dt = 0.0; }// Either none or double for hours over 50.

   // Time and a half.
   if (hours > 40)
      { ht = 1.5 * rate * (float)(hours - 40); }

   // Regular Pay
   if (hours >= 40)
     { rp = 40 * rate; }
   else
     { rp = (float)hours * rate; }

   pay = dt + ht + rp;   // Add three components of payroll.

   printf("\nThe pay is %.2f", pay);
   return 0;
}
```

4. The block of statements following the if can contain any valid C++ statement—even another `if` statement! This sometimes is handy, as the following example shows.

   You can even use this program to award employees for their years of service to your company. In this example, you are giving a gold watch to those with more than 20 years of service, a paperweight to those with more than 10 years, and a pat on the back to everyone else!

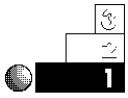
```
// Filename: C9SERV.CPP
// Prints a message depending on years of service.
#include <iostream.h>
main()
{
   int yrs;
   cout << "How many years of service? ";
   cin >> yrs;   // Determine the years they have worked.
```

```
   if (yrs > 20)
      { cout << "Give a gold watch\n"; }
   else
      { if (yrs > 10)
        { cout << "Give a paper weight\n"; }
        else
           { cout << "Give a pat on the back\n"; }
      }
   return 0;
}
```

Don't rely on the `if` within an `if` to handle too many conditions, because more than three or four conditions can add confusion. You might mess up your logic, such as: "If this is True, and if this is also True, then do something; but if not that, but something else is True, then..." (and so on). The `switch` statement that you learn about in a later chapter handles these types of multiple `if` selections much better than a long `if` within an `if` statement does.

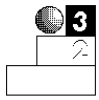
Review Questions

The answers to the review questions are in Appendix B.



1. Which operator tests for equality?
2. State whether each of these relational tests is True or False:
   a. `4 >= 5`
   b. `4 == 4`
   c. `165 >= 165`
   d. `0 != 25`
3. True or false: `C++ is fun` prints on-screen when the following statement executes.

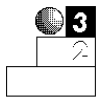
```
if (54 <= 54)
   { printf("C++ is fun"); }
```



4. What is the difference between an `if` and an `if-else` statement?

5. Does the following `printf()` execute?

```
if (3 != 4 != 1)
   { printf("This will print"); }
```



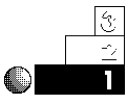
6. Using the ASCII table (see Appendix C, "ASCII Table"), state whether these character relational tests are True or False:

   a. `'C' < 'c'`

   b. `'0' > '0'`

   c. `'?' > ')'`

Review Exercises



1. Write a weather-calculator program that asks for a list of the previous five days' temperatures, then prints `Brrrr!` every time a temperature falls below freezing.

2. Write a program that asks for a number and then prints the square and cube (the number multiplied by itself three times) of the number you input, if that number is more than `1`. Otherwise, the program does not print anything.

3. In a program, ask the user for two numbers. Print a message telling how the first one relates to the second. In other words, if the user enters `5` and `7`, your program prints "`5 is less than 7`."

4. Write a program that prompts the user for an employee's pre-tax salary and prints the appropriate taxes. The taxes are 10 percent if the employee makes less than $10,000; 15 percent if the employee earns $10,000 up to, but not including, $20,000; and 20 percent if the employee earns $20,000 or more.

Summary

You now have the tools to write powerful data-checking programs. This chapter showed you how to compare literals, variables, and combinations of both by using the relational operators. The `if` and the `if-else` statements rely on such data comparisons to determine which code to execute next. You can now *conditionally execute* statements in your programs.

The next chapter takes this one step further by combining relational operators to create logical operators (sometimes called *compound conditions*). These logical operators further improve your program's capability to make selections based on data comparisons.

Logical Operators

C++'s *logical operators* enable you to combine relational operators into more powerful data-testing statements. The logical operators are sometimes called *compound relational operators.* As C++'s precedence table shows, relational operators take precedence over logical operators when you combine them. The precedence table plays an important role in these types of operators, as this chapter emphasizes.

This chapter introduces you to

- The logical operators
- How logical operators are used
- How logical operators take precedence

This chapter concludes your study of the conditional testing that C++ enables you to perform, and it illustrates many examples of `if` statements in programs that work on compound conditional tests.

Defining Logical Operators

There may be times when you have to test more than one set of variables. You can combine more than one relational test into a *compound relational test* by using C++'s logical operators, as shown in Table 10.1.

Table 10.1. Logical operators.

Operator	*Meaning*
&&	AND
||	OR
!	NOT

Logical operators enable the user to compute compound relational tests.

The first two logical operators, && and ||, never appear by themselves. They typically go between two or more relational tests.

Table 10.2 shows you how each logical operator works. These tables are called *truth tables* because they show you how to achieve True results from an if statement that uses these operators. Take some time to study these tables.

Table 10.2. Truth tables.

The AND (&&) truth table (Both sides must be True)		
True	AND	True = True
True	AND	False = False
False	AND	True = False
False	AND	False = False

The OR (||) truth table (One or the other side must be True)		
True	OR	True = True
True	OR	False = True
False	OR	True = True
False	OR	False = False

The NOT (!) truth table (Causes an opposite relation)	
NOT	True = False
NOT	False = True

Logical Operators and Their Uses

The True and False on each side of the operators represent a relational `if` test. The following statements, for example, are valid `if` tests that use logical operators (sometimes called *compound relational operators*).

If the variable `a` *is less than the variable* `b`*, and the variable* `c` *is greater than the variable* `d`*, then print* `Results are invalid.` *to the screen.*

```
if ((a < b) && (c > d))
   { cout << "Results are invalid."; }
```

The variable `a` must be less than `b` and, at the same time, `c` must be greater than `d` for the `printf()` to execute. The `if` statement still requires parentheses around its complete conditional test. Consider this portion of a program:

```
if ((sales > 5000) || (hrs_worked > 81))
   { bonus=500; }
```

The || is sometimes called *inclusive OR*. Here is a program segment that includes the not (!) operator:

The `sales` must be more than 5000, or the `hrs_worked` must be more than 81, before the assignment executes.

```
if (!(sales < 2500))
   { bonus = 500; }
```

If `sales` is greater than or equal to 2500, `bonus` is initialized. This illustrates an important programming tip: Use `!` sparingly. Or, as some professionals so wisely put it: "Do not use `!` or your programs will not be `!`(unclear)." It is much clearer to rewrite the previous example by turning it into a positive relational test:

```
if (sales >= 2500)
  { bonus 500; }
```

But the `!` operator is sometimes helpful, especially when testing for end-of-file conditions for disk files, as you learn in Chapter 30, "Sequential Files." Most the time, however, you can avoid using `!` by using the reverse logic shown in the following:

`!(var1 == var2)` is the same as `(var1 != var2)`

`!(var1 <= var2)` is the same as `(var1 > var2)`

`!(var1 >= var2)` is the same as `(var1 < var2)`

`!(var1 != var2)` is the same as `(var1 == var2)`

`!(var1 > var2)` is the same as `(var1 <= var2)`

`!(var1 < var2)` is the same as `(var1 >= var2)`

Notice that the overall format of the `if` statement is retained when you use logical operators, but the relational test expands to include more than one relation. You even can have three or more, as in the following statement:

```
if ((a == B) && (d == f) || (l = m) || !(k <> 2)) ...
```

This is a little too much, however, and good programming practice dictates using *at most* two relational tests inside a single `if` statement. If you have to combine more than two, use more than one `if` statement to do so.

As with other relational operators, you also use the following logical operators in everyday conversation.

> "If my pay is high and my vacation time is long, we can go to Italy this summer."
>
> "If you take the trash out or clean your room, you can watch TV tonight."
>
> "If you aren't good, you'll be punished."

Internal Truths

The True or False results of relational tests occur internally at the bit level. For example, take the `if` test:

```
if (a == 6) ...
```

to determine the truth of the relation, (`a==6`). The computer takes a binary 6, or 00000110, and compares it, bit-by-bit, to the variable `a`. If `a` contains 7, a binary 00000111, the result of this *equal* test is False, because the right bit (called the *least-significant bit*) is different.

C++'s Logical Efficiency

C++ attempts to be more efficient than other languages. If you combine multiple relational tests with one of the logical operators, C++ does not always interpret the full expression. This ultimately makes your programs run faster, but there are dangers! For example, if your program is given the conditional test:

```
if ((5 > 4) || (sales < 15) && (15 != 15))...
```

C++ only evaluates the first condition, `(5 > 4)`, and realizes it does not have to look further. Because `(5 > 4)` is True and because `||` (OR) anything that follows it is still True, C++ does not bother with the rest of the expression. The same holds true for the following statement:

```
if ((7 < 3) && (age > 15) && (initial == 'D'))...
```

Here, C++ evaluates only the first condition, which is False. Because the `&&` (AND) anything else that follows it is also False, C++ does not interpret the expression to the right of `(7 < 3)`. Most of the time, this doesn't pose a problem, but be aware that the following expression might not fulfill your expectations:

```
if ((5 > 4) || (num = 0))...
```

The `(num = 0)` assignment never executes, because C++ has to interpret only `(5 > 4)` to determine whether the entire expression is True or False. Due to this danger, do not include assignment expressions in the same condition as a logical test. The following single `if` condition:

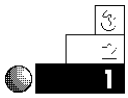
```
if ((sales > old_sales) || (inventory_flag = 'Y'))...
```

should be broken into two statements, such as:

```
inventory_flag) = 'Y';
if ((sales > old_sales) || (inventory_flag))...
```

so the `inventory_flag` is always assigned the `'Y'` value, no matter how the `(sales > old_sales)` expression tests.

Examples



1. The summer Olympics are held every four years during each year that is divisible evenly by 4. The U.S. Census is taken every 10 years, in each year that is evenly divisible by 10. The following short program asks for a year, and then tells the user if it is a year of the summer Olympics, a year of the census, or both. It uses relational operators, logical operators, and the modulus operator to determine this output.

```
// Filename: C10YEAR.CPP
// Determines if it is Summer Olympics year,
// U.S. Census year, or both.
#include <iostream.h>
main()
{
   int year;
   // Ask for a year
   cout << "What is a year for the test? ";
   cin >> year;

   // Test the year
   if (((year % 4)==0) && ((year % 10)==0))
     { cout << "Both Olympics and U.S. Census!";
       return 0; }  // Quit program, return to operating
                    // system.
   if ((year % 4)==0)
     { cout << "Summer Olympics only"; }
   else
      { if ((year % 10)==0)
         { cout << "U.S. Census only"; }
      }
   return 0;
}
```

2. Now that you know about compound relations, you can write an age-checking program like the one called C9AGE.CPP presented in Chapter 9, "Relational Operators." That program ensured the age would be above 10. This is another way you can validate input for reasonableness.

The following program includes a logical operator in its `if` to determine whether the age is greater than 10 and less than 100. If either of these is the case, the program concludes that the user did not enter a valid age.

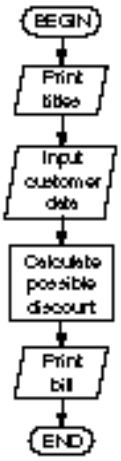
```
// Filename: C10AGE.CPP
// Program that helps ensure age values are reasonable.
#include <iostream.h>
main()
{
   int age;

   cout << "What is your age? ";
   cin >> age;
   if ((age < 10) || (age > 100))
     { cout << " \x07 \x07 \n";   // Beep twice
       cout << "*** The age must be between 10 and"
                    "100 ***\n"; }
   else
      { cout << "You entered a valid age."; }
   return 0;
}
```

3. The following program could be used by a video store to calculate a discount, based on the number of rentals people transact as well as their customer status. Customers are classified either `R` for *Regular* or `S` for *Special*. Special customers have been members of the rental club for more than one year. They automatically receive a 50-cent discount on all rentals. The store also holds "value days" several times a year. On value days, all customers receive the 50-cent discount. Special customers do not receive an additional 50 cents off during value days, because every day is a discount for them.

   The program asks for each customer's status and whether or not it is a value day. It then uses the `||` relation to test for the discount. Even before you started learning C++, you would probably have looked at this problem with the following idea in mind.

"If a customer is Special or if it is a value day, deduct 50 cents from the rental."

That's basically the idea of the `if` decision in the following program. Even though Special customers do not receive an additional discount on value days, there is one final `if` test for them that prints an extra message at the bottom of the screen's indicated billing.




```
// Filename: C10VIDEO.CPP
// Program that computes video rental amounts and gives
// appropriate discounts based on the day or customer status.
#include <iostream.h>
#include <stdio.h>
main()
{
   float tape_charge, discount, rental_amt;
   char first_name[15];
   char last_name[15];
   int num_tapes;
   char val_day, sp_stat;

   cout << "\n\n *** Video Rental Computation ***\n";
   cout << "     ------------------------\n";
   // Underline title

   tape_charge = 2.00;
   // Before-discount tape fee-per tape.

   // Receive input data.
   cout << "\nWhat is customer's first name? ";
   cin >> first_name;
   cout << "What is customer's last name? ";
   cin >> last_name;

   cout << "\nHow many tapes are being rented? ";
   cin >> num_tapes;

   cout << "Is this a Value day (Y/N)? ";
   cin >> val_day;

   cout << "Is this a Special Status customer (Y/N)? ";
   cin >> sp_stat;
   // Calculate rental amount.
```

```
 discount = 0.0;   // Increase discount if they are eligible.
   if ((val_day == 'Y') || (sp_stat == 'Y'))
     { discount = 0.5;
       rental_amt=(num_tapes*tape_charge)
                  (discount*num_tapes); }

   // Print the bill.
   cout << "\n\n** Rental Club **\n\n";
   cout << first_name << " " << last_name << " rented "
        << num_tapes << " tapes\n";
   printf("The total was %.2f\n", rental_amt);
   printf("The discount was %.2f per tape\n", discount);
   // Print extra message for Special Status customers.
   if (sp_stat == 'Y')
      { cout << "\nThank them for being a Special "
             << "Status customer\n";}
   return 0;
}
```

The output of this program appears below. Notice that Special customers have the extra message at the bottom of the screen. This program, due to its `if` statements, performs differently depending on the data entered. No discount is applied for Regular customers on nonvalue days.

```
*** Video Rental Computation ***
    ------------------------

What is customer's first name? Jerry
What is customer's last name? Parker

How many tapes are being rented? 3
Is this a Value day (Y/N)? Y
Is this a Special Status customer (Y/N)? Y

** Rental Club **

Jerry Parker rented 3 tapes
The total was 4.50
The discount was 0.50 per tape

Thank them for being a Special Status customer
```

Logical Operators and Their Precedence

The math precedence order you read about in Chapter 8, "Using C++ Math Operators and Precedence," did not include the logical operators. To be complete, you should be familiar with the entire order of precedence, as presented in Appendix D, "C++ Precedence Table."

You might wonder why the relational and logical operators are included in a precedence table. The following statement helps show you why:

```
if ((sales < min_sal * 2 && yrs_emp > 10 * sub) ...
```

Without the complete order of operators, it is impossible to determine how such a statement would execute. According to the precedence order, this `if` statement executes as follows:

```
if ((sales < (min_sal * 2)) && (yrs_emp > (10 * sub))) ...
```

This still might be confusing, but it is less so. The two multiplications are performed first, followed by the relations `<` and `>`. The `&&` is performed last because it is lowest in the precedence order of operators.

To avoid such ambiguous problems, be sure to use ample parentheses—even if the default precedence order is your intention. It is also wise to resist combining too many expressions inside a single `if` relational test.

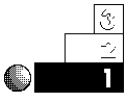
Notice that `||` (OR) has lower precedence than `&&` (AND). Therefore, the following `if` tests are equivalent:

```
if ((first_initial=='A') && (last_initial=='G') || (id==321)) ...
if (((first_initial=='A') && (last_initial=='G')) || (id==321)) ...
```

The second is clearer, due to the parentheses, but the precedence table makes them identical.

Review Questions

The answers to the review questions are in Appendix B.



1. What are the three logical operators?

2. The following compound relational tests produce True or False comparisons. Determine which are True and which are False.

   a. `! (True || False)`

   b. `(True && False) && (False || True)`

   c. `! (True && False)`

   d. `True || (False && False) || False`

3. Given the statement:

   ```
   int i=12, j=10, k=5;
   ```

   What are the results (True or False) of the following statements? (*Hint:* Remember that C++ interprets any nonzero statement as True.)

   a. `i && j`

   b. `12 - i || k`

   c. `j != k && i != k`

4. What is the value printed in the following program? (*Hint:* Don't be misled by the assignment operators on each side of the `||`.)

```
// Filename: C10LOGO.CPP
// Logical operator test
#include <iostream.h>
main()
{
   int f, g;

   g = 5;
   f = 8;
   if ((g = 25) || (f = 35))
```

```
      { cout << "g is " << g << " and f got changed to " << f; }
   return 0;
}
```

5. Using the precedence table, determine whether the following statements produce a True or False result. After this, you should appreciate the abundant use of parentheses!

   a. `5 == 4 + 1 || 7 * 2 != 12 - 1 && 5 == 8 / 2`

   b. `8 + 9 != 6 - 1 || 10 % 2 != 5 + 0`

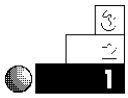
   c. `17 - 1 > 15 + 1 && 0 + 2 != 1 == 1 || 4 != 1`

   d. `409 * 0 != 1 * 409 + 0 || 1 + 8 * 2 >= 17`

6. Does the following `cout` execute?

```
if (!0)
   { cout << "C++ By Example \n"; }
```

Review Exercises



1. Write a program (by using a single compound `if` statement) to determine whether the user enters an odd positive number.

2. Write a program that asks the user for two initials. Print a message telling the user if the first initial falls alphabetically before the second.

3. Write a number-guessing game. Assign a value to a variable called `number` at the top of the program. Give a prompt that asks for five guesses. Receive the user's five guesses with a single `scanf()` for practice with `scanf()`. Determine whether any of the guesses match the `number` and print an appropriate message if one does.

4. Write a tax-calculation routine, as follows: A family pays no tax if its income is less than $5,000. It pays a 10 percent tax if its income is $5,000 to $9,999, inclusive. It pays a 20 percent tax if the income is $10,000 to $19,999, inclusive. Otherwise, it pays a 30 percent tax.

Summary

This chapter extended the `if` statement to include the `&&`, `||`, and `!` logical operators. These operators enable you to combine several relational tests into a single test. C++ does not always have to look at every relational operator when you combine them in an expression.

This chapter concludes the explanation of the `if` statement. The next chapter explains the remaining regular C++ operators. As you saw in this chapter, the precedence table is still important to the C++ language. Whenever you are evaluating expressions, keep the precedence table in the back of your mind (or at your fingertips) at all times!

Additional C++ Operators

C++ has several other operators you should learn besides those you learned in Chapters 9 and 10. In fact, C++ has more operators than most programming languages. Unless you become familiar with them, you might think C++ programs are cryptic and difficult to follow. C++'s heavy reliance on its operators and operator precedence produces the efficiency that enables your programs to run more smoothly and quickly.

This chapter teaches you the following:

- The `?:` conditional operator
- The `++` increment operator
- The `--` decrement operator
- The `sizeof` operator
- The `(,)` comma operator
- The Bitwise Operators (`&`, `|`, and `^`)

Most the operators described in this chapter are unlike those found in any other programming language. Even if you have programmed in other languages for many years, you still will be surprised by the power of these C++ operators.

The Conditional Operator

The conditional operator is a ternary operator.

The *conditional operator* is C++'s only *ternary* operator, requiring three operands (as opposed to the unary's single-and the binary's double-operand requirements). The conditional operator is used to replace `if-else` logic in some situations. The conditional operator is a two-part symbol, `?:`, with a format as follows:

```
conditional_expression ? expression1 : expression2;
```

The `conditional_expression` is any expression in C++ that results in a True (nonzero) or False (zero) answer. If the result of `conditional_expression` is True, `expression1` executes. Otherwise, if the result of `conditional_expression` is False, `expression2` executes. Only one of the expressions following the question mark ever executes. Only a single semicolon appears at the end of `expression2`. The internal expressions, such as `expression1`, do not have a semicolon. Figure 11.1 illustrates the conditional operator more clearly.

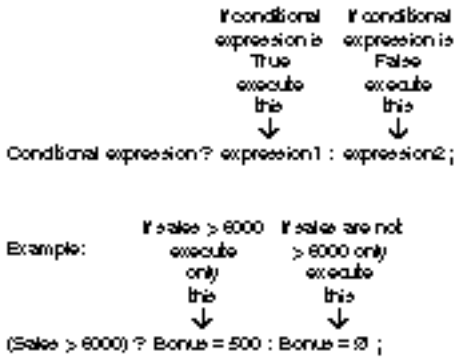


Figure 11.1. Format of the conditional operator.

If you require simple if-else logic, the conditional operator usually provides a more direct and succinct method, although you should always prefer readability over compact code.

To glimpse the conditional operator at work, consider the section of code that follows.

```
if (a > b)
   { ans = 10; }
else
   { ans = 25; }
```

You can easily rewrite this kind of if-else code by using a single conditional operator.

If the variable a *is greater than the variable* b*, make the variable* ans *equal to 10; otherwise, make* ans *equal to 25.*

```
a > b ? (ans = 10) : (ans = 25);
```

Although parentheses are not required around conditional_expression to make it work, they usually improve readability. This statement's readability is improved by using parentheses, as follows:

```
(a > b) ? (ans = 10) : (ans = 25);
```

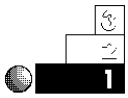
Because each C++ expression has a value—in this case, the value being assigned—this statement could be even more succinct, without loss of readability, by assigning ans the answer to the left of the conditional:

```
ans = (a > b) ? (10) : (25);
```

This expression says: If a is greater than b, assign 10 to ans; otherwise, assign 25 to ans. Almost any if-else statement can be rewritten as a conditional, and vice versa. You should practice converting one to the other to familiarize yourself with the conditional operator's purpose.

NOTE: Any valid if C++ statement also can be a conditional_expression, including all relational and logical operators as well as any of their possible combinations.

Examples



1. Suppose you are looking over your early C++ programs, and you notice the following section of code.

```
if (production > target)
   { target *= 1.10; }
else
   { target *= .90; }
```

   You should realize that such a simple `if-else` statement can be rewritten using a conditional operator, and that more efficient code results. You can therefore change it to the following single statement.

```
(production > target) ? (target *= 1.10) : (target *= .90);
```

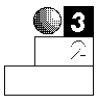
2. Using a conditional operator, you can write a routine to find the minimum value between two variables. This is sometimes called a *minimum routine*. The statement to do this is

```
minimum = (var1 < var2) ? var1 : var2;
```

   If `var1` is less than `var2`, the value of `var1` is assigned to `minimum`. If `var2` is less, the value of `var2` is assigned to `minimum`. If the variables are equal, the value of `var2` is assigned to `minimum`, because it does not matter which is assigned.

3. A *maximum routine* can be written just as easily:

```
maximum = (var1 > var2) ? var1 : var2;
```



4. Taking the previous examples a step further, you can also test for the sign of a variable. The following conditional expression assigns –1 to the variable called `sign` if `testvar` is less than 0; 0 to `sign` if `testvar` is zero; and +1 to `sign` if `testvar` is 1 or more.

```
sign = (testvar < 0) ? -1 : (testvar > 0);
```

   It might be easy to spot why the less-than test results in a –1, but the second part of the expression can be confusing. This works well due to C++'s 1 and 0 (for True and False, respectively) return values from a relational test. If `testvar` is 0 or greater, `sign` is assigned the answer `(testvar > 0)`. The value

of `(testvar > 0)` is 1 if True (therefore, `testvar` is more than 0) or 0 if `testvar` is equal to 0.

The preceding statement shows C++'s efficient conditional operator. It might also help you understand if you write the statement using typical `if-else` logic. Here is the same problem written with a typical `if-else` statement:

```
if (testvar < 0)
   { sign = -1; }
else
   { sign = (testvar > 0); }    // testvar can only be
                                // 0 or more here.
```

The Increment and Decrement Operators

The ++ operator adds 1 to a variable. The –– operator subtracts 1 from a variable.

C++ offers two unique operators that add or subtract 1 to or from variables. These are the *increment* and *decrement* operators: ++ and ––. Table 11.1 shows how these operators relate to other types of expressions you have seen. Notice that the ++ and –– can appear on either side of the modified variable. If the ++ or –– appears on the left, it is known as a *prefix* operator. If the operator appears on the right, it is a *postfix* operator.

Table 11.1. The ++ and –– operators.

Operator	*Example*	*Description*	*Equivalent Statements*	
++	`i++;`	postfix	`i = i + 1;`	`i += 1;`
++	`++i;`	prefix	`i = i + 1;`	`i += 1;`
––	`i––;`	postfix	`i = i - 1;`	`i -= 1;`
––	`––i;`	prefix	`i = i - 1;`	`i -= 1;`

Any time you have to add 1 or subtract 1 from a variable, you can use these two operators. As Table 11.1 shows, if you have to increment or decrement only a single variable, these operators enable you to do so.

Increment and Decrement Efficiency

The increment and decrement operators are straightforward, efficient methods for adding 1 to a variable and subtracting 1 from a variable. You often have to do this during counting or processing loops, as discussed in Chapter 12, "The `while` Loop" and beyond.

These two operators compile directly into their assembly language equivalents. Almost all computers include, at their lowest binary machine-language commands, increment and decrement instructions. If you use C++'s increment and decrement operators, you ensure that they compile to these low-level equivalents.

If, however, you code expressions to add or subtract 1 (as you do in other programming languages), such as the expression `i = i - 1`, you do not actually ensure that C++ compiles this instruction in its efficient machine-language equivalent.

Whether you use prefix or postfix does not matter—if you are incrementing or decrementing single variables on lines by themselves. However, when you combine these two operators with other operators in a single expression, you must be aware of their differences. Consider the following program section. Here, all variables are integers because the increment and decrement operators work only on integer variables.

Make `a` *equal to 6. Increment* `a`*, subtract 1 from it, then assign the result to* `b`*.*

```
a = 6;
b = ++a - 1;
```

What are the values of `a` and `b` after these two statements finish? The value of `a` is easy to determine: it is incremented in the second statement, so it is 7. However, `b` is either 5 or 6 depending on when the variable `a` increments. To determine when `a` increments, consider the following rule:

- If a variable is incremented or decremented with a *prefix* operator, the increment or decrement occurs *before* the variable's value is used in the remainder of the expression.
- If a variable is incremented or decremented with a *postfix* operator, the increment or decrement occurs *after* the variable's value is used in the remainder of the expression.

In the previous code, `a` contains a prefix increment. Therefore, its value is first incremented to 7, then 1 is subtracted from 7, and the result (6) is assigned to `b`. If a postfix increment is used, as in

```
a = 6;
b = a++ - 1;
```

`a` is 6, therefore, 5 is assigned to `b` because `a` does not increment to 7 until after its value is used in the expression. The precedence table in Appendix D, "C++ Precedence Table," shows that prefix operators contain much higher precedence than almost every other operator, especially low-precedence postfix increments and decrements.

TIP: If the order of prefix and postfix confuses you, break your expressions into two lines of code and type the increment or decrement before or after the expression that uses it.

By taking advantage of this tip, you can now rewrite the previous example as follows:

```
a = 6;
b = a - 1;
a++;
```

There is now no doubt as to when `a` is incremented: `a` increments after `b` is assigned to `a-1`.

Even parentheses cannot override the postfix rule. Consider the following statement.

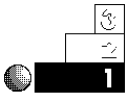
```
x = p + (((amt++)));
```

There are too many unneeded parentheses here, but even the redundant parentheses are not enough to increment `amt` before adding its value to `p`. Postfix increments and decrements *always* occur after their variables are used in the surrounding expression.

CAUTION: Do not attempt to increment or decrement an expression. You can apply these operators only to variables. The following expression is invalid:

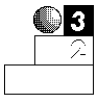
```
sales = ++(rate * hours);  // Not allowed!!
```

Examples



1. As you should with all other C++ operators, keep the precedence table in mind when you evaluate expressions that increment and decrement. Figures 11.2 and 11.3 show you some examples that illustrate these operators.

2. The precedence table takes on even more meaning when you see a section of code such as that shown in Figure 11.3.



3. Considering the precedence table—and, more importantly, what you know about C++'s relational efficiencies—what is the value of the `ans` in the following section of code?

```
int i=1, j=20, k=-1, l=0, m=1, n=0, o=2, p=1;
ans = i || j-- && k++ || ++l && ++m || n-- & !o || p--;
```

This, at first, seems to be extremely complicated. Nevertheless, you can simply glance at it and determine the value of `ans`, as well as the ending value of the rest of the variables.

Recall that when C++ performs a relation `||` (or), it ignores the right side of the `||` if the left value is True (any nonzero value is True). Because any nonzero value is True, C++ does

not evaluate the values on the right. Therefore, C++ performs this expression as shown:

```
ans = i || j-- && k++ || ++l && ++m || n-- & !o || p--;
          |
          1 (TRUE)
```

```
int i=1;
int j=2;
int k=3;
ans = i++ * j - --k;
                |
      i++ * j -   2
        |
        2   -   2
            |
            0

ans = 0, then i increments by 1 to its final value of 2.

int i=1;
int j=2;
int k=3;
ans = ++i * j - k--;
        |
       2  * j - k--
         |
         4    - k--
              |
              1

ans = 1, then k decrements by 1 to its final value of 2.
```

Figure 11.2. C++ operators incrementing (above) and decrementing (below) by order of precedence.

```
int i=0;
int j=-1;
int k=0;
int m=1
ans = i++ && ++j || k || m++;
                |
      i++ &&  0  || k || m++
          \  /
           0     || k || m++
             \  /
              0      || m++
                \  /
                 1

ans = 1, then i increments by 1 to its final value of 1,
and m increments by 1 to its final value of 2.
```

Figure 11.3. Another example of C++ operators and their precedence.

NOTE: Because `i` is True, C++ evaluates the entire expression as True and ignores all code after the first `||`. Therefore, *every other increment and decrement expression is ignored.* Because C++ ignores the other expressions, only `ans` is changed by this expression. The other variables, `j` through `p`, are never incremented or decremented, even though several of them contain increment and decrement operators. If you use relational operators, be aware of this problem and break out all increment and decrement operators into statements by themselves, placing them on lines before the relational statements that use their values.

The **sizeof** Operator

There is another operator in C++ that does not look like an operator at all. It looks like a built-in function, but it is called the

sizeof operator. In fact, if you think of sizeof as a function call, you might not become confused because it works in a similar way. The format of sizeof follows:


```
sizeof data
```

or

```
sizeof(data type)
```

The sizeof operator is unary, because it operates on a single value. This operator produces a result that represents the size, in bytes, of the data or data type specified. Because most data types and variables require different amounts of internal storage on different computers, the sizeof operator enables programs to maintain consistency on different types of computers.

TIP: Most C++ programmers use parentheses around the sizeof argument, whether that argument is data or data type. Because you *must* use parentheses around data type arguments and you *can* use them around data arguments, it doesn't hurt to always use them.

The sizeof operator returns its argument's size in bytes.

The sizeof operator is sometimes called a *compile-time operator*. At compile time, rather than runtime, the compiler replaces each occurrence of sizeof in your program with an unsigned integer value. Because sizeof is used more in advanced C++ programming, this operator is better utilized later in the book for performing more advanced programming requirements.

If you use an array as the sizeof argument, C++ returns the number of bytes you originally reserved for that array. Data inside the array have nothing to do with its returned sizeof value—even if it's only a character array containing a short string.

Examples

1. Suppose you want to know the size, in bytes, of floating-point variables for your computer. You can determine this by entering the keyword float in parentheses—after sizeof—as shown in the following program.

```
// Filename: C11SIZE1.CPP
// Prints the size of floating-point values.
#include <iostream.h>
main()
{
   cout << "The size of floating-point variables on \n";
   cout << "this computer is " << sizeof(float) << "\n";
   return 0;
}
```

This program might produce different results on different computers. You can use any valid data type as the `sizeof` argument. On most PCs, this program probably produces this output:

```
The size of floating-point variables on
this computer is: 4
```

The Comma Operator

Another C++ operator, sometimes called a *sequence point*, works a little differently. This is the *comma operator* (`,`), which does not directly operate on data, but produces a left-to-right evaluation of expressions. This operator enables you to put more than one expression on a single line by separating each one with a comma.

You already saw one use of the sequence point comma when you learned how to declare and initialize variables. In the following section of code, the comma separates statements. Because the comma associates from the left, the first variable, `i`, is declared and initialized before the second variable.

```
main()
{
   int i=10, j=25;
   // Remainder of the program follows.
```

However, the comma is *not* a sequence point when it is used inside function parentheses. Then it is said to *separate* arguments, but it is not a sequence point. Consider the `printf()` that follows.

```
printf("%d %d %d", i, i++, ++i);
```

Many results are possible from such a statement. The commas serve only to separate arguments of the `printf()`, and do not generate the left-to-right sequence that they otherwise do when they aren't used in functions. With the statement shown here, you are not ensured of *any* order! The postfix `i++` might possibly be performed before the prefix `++i`, even though the precedence table does not require this. Here, the order of evaluation depends on how your compiler sends these arguments to the `printf()` function.

TIP: Do not put increment operators or decrement operators in function calls because you cannot predict the order in which they execute.

Examples

1. You can put more than one expression on a line, using the comma as a sequence point. The following program does this.

```
// Filename: C11COM1.CPP
// Illustrates the sequence point.
#include <iostream.h>
main()
{
   int num, sq, cube;
   num = 5;

   // Calculate the square and cube of the number.
   sq = (num * num), cube = (num * num * num);

   cout << "The square of " << num << " is " << sq <<
           " and the cube is " << cube;
   return 0;
}
```

This is not necessarily recommended, however, because it doesn't add anything to the program and actually decreases its readability. In this example, the square and cube are probably better computed on two separate lines.

2. The comma enables some interesting statements. Consider the following section of code.

```
i = 10
j = (i = 12, i + 8);
```

When this code finishes executing, `j` has the value of 20—even though this is not necessarily clear. In the first statement, `i` is assigned 10. In the second statement, the comma causes `i` to be assigned a value of 12, then `j` is assigned the value of `i + 8`, or 20.

3. In the following section of code, `ans` is assigned the value of 12, because the assignment *before* the comma is performed first. Despite this right-to-left associativity of the assignment operator, the comma's sequence point forces the assignment of 12 to `x` before `x` is assigned to `ans`.

```
ans = (y = 8, x = 12);
```

When this fragment finishes, `y` contains 8, `x` contains 12, and `ans` also contains 12.

Bitwise Operators

The *bitwise operators* manipulate internal representations of data and not just "values in variables" as the other operators do. These bitwise operators require an understanding of Appendix A's binary numbering system, as well as a computer's memory. This section introduces the bitwise operators. The bitwise operators are used for advanced programming techniques and are generally used in much more complicated programs than this book covers.

Some people program in C++ for years and never learn the bitwise operators. Nevertheless, understanding them can help you improve a program's efficiency and enable you to operate at a more advanced level than many other programming languages allow.

Bitwise Logical Operators

There are four bitwise logical operators, and they are shown in Table 11.2. These operators work on the binary representations of integer data. This enables systems programmers to manipulate internal bits in memory and in variables. The bitwise operators are not just for systems programmers, however. Application programmers also can improve their programs' efficiency in several ways.

Table 11.2. Bitwise logical operators.

Operator	*Meaning*
&	Bitwise AND
|	Bitwise inclusive OR
^	Bitwise exclusive OR
~	Bitwise 1's complement

Bitwise operators make bit-by-bit comparisons of internal data.

Each of the bitwise operators makes a bit-by-bit comparison of internal data. Bitwise operators apply only to character and integer variables and constants, and not to floating-point data. Because binary numbers consist of 1s and 0s, these 1s and 0s (called *bits*) are compared to each other to produce the desired result for each bitwise operator.

Before you study the examples, you should understand Table 11.3. It contains truth tables that describe the action of each bitwise operator on an integer's—or character's—internal-bit patterns.

Table 11.3. Truth tables.

Bitwise AND (&)
0 & 0 = 0
0 & 1 = 0
1 & 0 = 0
1 & 1 = 1

continues

Table 11.3. Continued.

Bitwise inclusive OR (|)
`0 | 0 = 0`
`0 | 1 = 1`
`1 | 0 = 1`
`1 | 1 = 1`
Bitwise exclusive OR (^)
`0 ^ 0 = 0`
`0 ^ 1 = 1`
`1 ^ 0 = 1`
`1 ^ 1 = 0`
Bitwise 1's complement (~)
`~0 = 1`
`~1 = 0`

In bitwise truth tables, you can replace the 1 and 0 with True and False, respectively, if it helps you to understand the result better. For the bitwise AND (`&`) truth table, both bits being compared by the `&` operator must be True for the result to be True. In other words, "True AND True results in True."

TIP: By replacing the 1s and 0s with True and False, you might be able to relate the bitwise operators to the regular logical operators, `&&` and `||`, that you use for `if` comparisons.

For bitwise ^, one side *or* the other—but not both—must be 1.

The `|` bitwise operator is sometimes called the *bitwise inclusive OR* operator. If one side of the `|` operator is 1 (True)—or if both sides are 1—the result is 1 (True).

The `^` operator is called *bitwise exclusive OR*. It means that either side of the `^` operator must be 1 (True) for the result to be 1 (True), but both sides cannot be 1 (True) at the same time.

The ~ operator, called *bitwise 1's complement*, reverses each bit to its opposite value.

NOTE: Bitwise 1's complement does *not* negate a number. As Appendix A, "Memory Addressing, Binary, and Hexadecimal Review," shows, most computers use a 2's complement to negate numbers. The bitwise 1's complement reverses the bit pattern of numbers, but it doesn't add the additional 1 as the 2's complement requires.

You can test and change individual bits inside variables to check for patterns of data. The following examples help to illustrate each of the four bitwise operators.

Examples

1. If you apply the bitwise & operator to numerals 9 and 14, you receive a result of 8. Figure 11.4 shows you why this is so. When the binary values of 9 (1001) and 14 (1110) are compared on a bitwise & basis, the resulting bit pattern is 8 (1000).

```
  1  0  0  1  (9)
  ↓  ↓  ↓  ↓
  &  &  &  &
  1  1  1  0  (14)
 ---------------
= 1  0  0  0  (8)
```

Figure 11.4. Performing bitwise & on 9 and 14.

In a C++ program, you can code this bitwise comparison as follows.

Make `result` *equal to the binary value of 9 (1001) ANDed to the binary value of 14 (1110).*

```
result = 9 & 14;
```

The `result` variable holds 8, which is the result of the bitwise `&`. The 9 (binary 1001) or 14 (binary 1110)—or both—also can be stored in variables with the same result.

2. When you apply the bitwise `|` operator to the numbers 9 and 14, you get 15. When the binary values of 9 (1001) and 14 (1110) are compared on a bitwise `|` basis, the resulting bit pattern is 15 (1111). `result`'s bits are 1 (True) in every position where a 1 appears in both numbers.

   In a C++ program, you can code this bitwise comparison as follows:

   ```
   result = 9 | 14;
   ```

   The `result` variable holds 15, which is the result of the bitwise `|`. The 9 or 14 (or both) also can be stored in variables.

3. The bitwise `^` applied to 9 and 14 produces 7. Bitwise `^` sets the resulting bits to 1 if one number or the other's bit is 1, but not if both of the matching bits are 1 at the same time.

   In a C++ program, you can code this bitwise comparison as follows:

   ```
   result = 9 ^ 14;
   ```

   The `result` variable holds 7 (binary 0111), which is the result of the bitwise `^`. The 9 or 14 (or both) also can be stored in variables with the same result.

4. The bitwise `~` simply negates each bit. It is a unary bitwise operator because you can apply it to only a single value at any one time. The bitwise `~` applied to 9 results in 6, as shown in Figure 11.5.

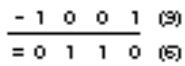


Figure 11.5. Performing bitwise ~ on the number 9.

In a C++ program, you can code this bitwise operation like this:

```
result = ~9;
```

The `result` variable holds 6, which is the result of the bitwise ~. The 9 can be stored in a variable with the same result.

5. You can take advantage of the bitwise operators to perform tests on data that you cannot do as efficiently in other ways.

   For example, suppose you want to know if the user typed an odd or even number (assuming integers are being input). You can use the modulus operator (`%`) to determine whether the remainder—after dividing the input value by 2—is 0 or 1. If the remainder is 0, the number is even. If the remainder is 1, the number is odd.

   The bitwise operators are more efficient than other operators because they directly compare bit patterns without using any mathematical operations.

   Because a number is even if its bit pattern ends in a 0 and odd if its bit pattern ends in 1, you also can test for odd or even numbers by applying the bitwise `&` to the data and to a binary 1. This is more efficient than using the modulus operator. The following program informs users if their input value is odd or even using this technique.

Identify the file and include the input/output header file. This program tests for odd or even input. You need a place to put the user's number, so declare the `input` *variable as an integer.*

Ask the user for the number to be tested. Put the user's answer in `input`*. Use the bitwise operator,* `&`*, to test the number. If the bit on the extreme right in* `input` *is 1, tell the user that the number is odd. If the bit on the extreme right in* `input` *is 0, tell the user that the number is even.*


```
// Filename: C11ODEV.CPP
// Uses a bitwise & to determine whether a
// number is odd or even.
#include <iostream.h>
main()
{
```

```
    int input;                    // Will hold user's number
    cout << "What number do you want me to test? ";
    cin >> input;

    if (input & 1)              // True if result is 1;
                                // otherwise it is false (0)
       { cout << "The number " << input << " is odd\n"; }
    else
       { cout << "The number " << input << " is even\n"; }
    return 0;
}
```

6. The only difference between the bit patterns for uppercase and lowercase characters is bit number 5 (the third bit from the left, as shown in Appendix A, "Memory Addressing, Binary, and Hexadecimal Review"). For lowercase letters, bit 5 is a 1. For uppercase letters, bit 5 is a 0. Figure 11.6 shows how *A* and *B* differ from *a* and *b* by a single bit.

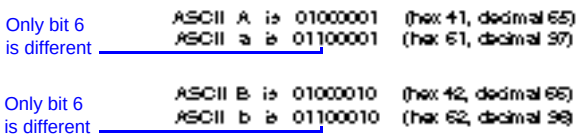


Figure 11.6. Bitwise difference between two uppercase and two lowercase ASCII letters.

To convert a character to uppercase, you have to turn off (change to a 0) bit number 5. You can apply a bitwise & to the input character and 223 (which is 11011111 in binary) to turn off bit 5 and convert any input character to its uppercase equivalent. If the number is already in uppercase, this bitwise & does not change it.

The 223 (binary 11011111) is called a *bit mask* because it masks (just as masking tape masks areas not to be painted) bit 5 so it becomes 0, if it is not already. The following program does this to ensure that users typed uppercase characters when they were asked for their initials.

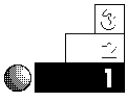
```
// Filename: C11UPCS1.CPP
// Converts the input characters to uppercase
// if they aren't already.
#include <iostream.h>
main()
{
   char first, middle, last;   // Will hold user's initials
   int bitmask=223;                  // 11011111 in binary

   cout << "What is your first initial? ";
   cin >> first;
   cout << "What is your middle initial? ";
   cin >> middle;
   cout << "What is your last initial? ";
   cin >> last;

   // Ensure that initials are in uppercase.
   first = first & bitmask;            // Turn off bit 5 if
   middle = middle & bitmask;          // it is not already
   last = last & bitmask;              // turned off.

   cout << "Your initials are " << first << " " <<
           middle << " " << last;
   return 0;
}
```

The following output shows what happens when two of the initials are typed with lowercase letters. The program converts them to uppercase before printing them again. Although there are other ways to convert to lowercase, none are as efficient as using the & bitwise operator.

```
What is your first initial? g
What is your middle initial? M
What is your last initial? p
Your initials are: G M P
```

Review Questions

The answers to the review questions are in Appendix B.



1. What set of statements does the conditional operator replace?
2. Why is the conditional operator called a “ternary” operator?

3. Rewrite the following conditional operator as an `if-else` statement.

```
ans = (a == b) ? c + 2 : c + 3;
```

4. True or false: The following statements produce the same results.

```
var++;
```

and

```
var = var + 1;
```

5. Why is using the increment and decrement operators more efficient than using the addition and subtraction operators?
6. What is a sequence point?
7. Can the output of the following code section be determined?

```
age = 20;
printf(“You are now %d, and will be %d in one year”,
        age, age++);
```

8. What is the output of the following program section?

```
char name[20] = “Mike”;
cout << “The size of name is “ << sizeof(name) << “\n”;
```

9. What is the result of each of the following bitwise True-False expressions?

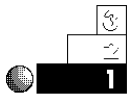
   a. `1 ^ 0 & 1 & 1 | 0`

   b. `1 & 1 & 1 & 1`

   c. `1 ^ 1 ^ 1 ^ 1`

   d. `~(1 ^ 0)`

Review Exercises



1. Write a program that prints the numerals from 1 to 10. Use ten different `cout`s and only one variable called `result` to hold the value before each `cout`. Use the increment operator to add 1 to `result` before each `cout`.

2. Write a program that asks users for their ages. Using a single `printf()` that includes a conditional operator, print on-screen the following if the input age is over 21,

   ```
   You are not a minor.
   ```

   or print this otherwise:

   ```
   You are still a minor.
   ```

   This `printf()` might be long, but it helps to illustrate how the conditional operator can work in statements where `if-else` logic does not.

3. Use the conditional operator—and no `if-else` statements—to write the following tax-calculation routine: A family pays no tax if its annual salary is less than $5,000. It pays a 10 percent tax if the salary range begins at $5,000 and ends at $9,999. It pays a 20 percent tax if the salary range begins at $10,000 and ends at $19,999. Otherwise, the family pays a 30 percent tax.

4. Write a program that converts an uppercase letter to a lowercase letter by applying a bitmask and one of the bit-wise logical operators. If the character is already in lowercase, do not change it.

Summary

Now you have learned almost every operator in the C++ language. As explained in this chapter, conditional, increment, and decrement are three operators that enable C++ to stand apart from many other programming languages. You must always be aware of the precedence table whenever you use these, as you must with all operators.

The `sizeof` and sequence point operators act unlike most others. The `sizeof` is a compile operator, and it works in a manner similar to the `#define` preprocessor directive because they are both replaced by their values at compile time. The sequence point enables you to have multiple statements on the same line—or in a single expression. Reserve the sequence point for declaring variables only because it can be unclear when it's combined with other expressions.

This chapter concludes the discussion on C++ operators. Now that you can compute just about any result you will ever need, it is time to discover how to gain more control over your programs. The next few chapters introduce control loops that give you repetitive power in C++.

The `while` Loop

The repetitive capabilities of computers make them good tools for processing large amounts of information. Chapters 12-15 introduce you to C++ constructs, which are the control and looping commands of programming languages. C++ constructs include powerful, but succinct and efficient, looping commands similar to those of other languages you already know.

The `while` loops enable your programs to repeat a series of statements, over and over, as long as a certain condition is always met. Computers do not get "bored" while performing the same tasks repeatedly. This is one reason why they are so important in business data processing.

This chapter teaches you the following:

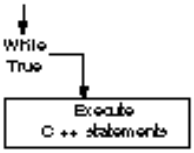
- The `while` loop
- The concept of loops
- The `do-while` loop
- Differences between `if` and `while` loops
- The `exit()` function
- The `break` statement
- Counters and totals

After completing this chapter, you should understand the first of several methods C++ provides for repeating program sections. This chapter's discussion of loops includes one of the most important uses for looping: creating counter and total variables.

The `while` Statement

The `while` statement is one of several C++ *construct statements.* Each construct (from *construction*) is a programming language statement—or a series of statements—that controls looping. The `while`, like other such statements, is a *looping statement* that controls the execution of a series of other statements. Looping statements cause parts of a program to execute repeatedly, as long as a certain condition is being met.

The format of the `while` statement is




```
while (test expression)
   { block of one or more C++ statements; }
```

The parentheses around `test expression` are required. As long as `test expression` is True (nonzero), the *block* of one or more C++ statements executes repeatedly until `test expression` becomes False (evaluates to zero). Braces are required before and after the body of the `while` loop, unless you want to execute only one statement. Each statement in the body of the `while` loop requires an ending semicolon.

The body of a `while` loop executes repeatedly as long as test expression is True.

The placeholder `test expression` usually contains relational, and possibly logical, operators. These operators provide the True-False condition checked in `test expression`. If `test expression` is False when the program reaches the `while` loop for the first time, the body of the `while` loop does not execute at all. Regardless of whether the body of the `while` loop executes no times, one time, or many times, the statements following the `while` loop's closing brace execute if `test expression` becomes False.

Because `test expression` determines when the loop finishes, the body of the `while` loop must change the variables used in `test expression`. Otherwise, `test expression` never changes and the `while` loop repeats forever. This is known as an *infinite loop,* and you should avoid it.

TIP: If the body of the `while` loop contains only one statement, the braces surrounding it are not required. It is a good habit to enclose all `while` loop statements in braces, however, because if you have to add statements to the body of the `while` loop later, your braces are already there.

The Concept of Loops

You use the loop concept in everyday life. Any time you have to repeat the same procedure, you are performing a loop—just as your computer does with the `while` statement. Suppose you are wrapping holiday gifts. The following statements represent the looping steps (in `while` format) that you follow while gift-wrapping.

```
while (there are still unwrapped gifts)
  { Get the next gift;
      Cut the wrapping paper;
      Wrap the gift;
      Put a bow on the gift;
      Fill out a name card for the gift;
      Put the wrapped gift with the others; }
```

Whether you have 3, 15, or 100 gifts to wrap, you use this procedure (loop) repeatedly until every gift is wrapped. For an example that is more easily computerized, suppose you want to total all the checks you wrote in the previous month. You could perform the following loop.

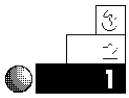
```
while (there are still checks from the last month to be totaled)
  { Add the amount of the next check to the total; }
```

The body of this pseudocode `while` loop has only one statement, but that single statement must be performed until you have added each one of the previous month's checks. When this loop ends (when no more checks from the previous month remain to be totaled), you have the result.

The body of a `while` loop can contain one or more C++ statements, including additional `while` loops. Your programs will be

more readable if you indent the body of a `while` loop a few spaces to the right. The following examples illustrate this.

Examples



1. Some programs presented earlier in the book require user input with `cin`. If users do not enter appropriate values, these programs display an error message and ask the user to enter another value, which is an acceptable procedure.

   Now that you understand the `while` loop construct, however, you should put the error message inside a loop. In this way, users see the message continually until they type proper input values, rather than once.

   The following program is short, but it demonstrates a `while` loop that ensures valid keyboard input. It asks users whether they want to continue. You can incorporate this program into a larger one that requires user permission to continue. Put a prompt, such as the one presented here, at the bottom of a text screen. The text remains on-screen until the user tells the program to continue executing.

Identify the file and include the necessary header file. In this program, you want to ensure the user enters Y or N. You have to store the user's answer, so declare the `ans` *variable as a character. Ask the users whether they want to continue, and get the response. If the user doesn't type Y or N, ask the user for another response.*


```
// Filename: C12WHIL1.CPP
// Input routine to ensure user types a
// correct response. This routine can be part
// of a larger program.
#include <iostream.h>
main()
{
   char ans;

   cout << "Do you want to continue (Y/N)? ";
   cin >> ans;                  // Get user's answer
```

```
   while ((ans != 'Y') && (ans != 'N'))
      { cout << "\nYou must type a Y or an N\n";  // Warn
                                                  // and ask
        cout << "Do you want to continue (Y/N)?"; // again.
        cin >> ans;
      }          // Body of while loop ends here.

   return 0;
}
```

Notice that the two `cin` functions do the same thing. You must use an initial `cin`, outside the `while` loop, to provide an answer for the `while` loop to check. If users type something other than Y or N, the program prints an error message, asks for another answer, then checks the new answer. This validation method is preferred over one where the reader only has one additional chance to succeed.

The `while` loop tests the test expression at the top of the loop. This is why the loop might never execute. If the test is initially False, the loop does not execute even once. The output from this program is shown as follows. The program repeats indefinitely, until the relational test is True (as soon as the user types either Y or N).

```
Do you want to continue (Y/N)? k

You must type a Y or an N
Do you want to continue (Y/N)? c

You must type a Y or an N
Do you want to continue (Y/N)? s

You must type a Y or an N
Do you want to continue (Y/N)? 5

You must type a Y or an N
Do you want to continue (Y/N)? Y
```

2. The following program is an example of an *invalid* `while` loop. See if you can find the problem.

```
// Filename: C12WHBAD.CPP
// Bad use of a while loop.
#include <iostream.h>
main()
{
   int a=10, b=20;
   while (a > 5)
      { cout << "a is " << a << ", and b is " << b << "\n";
        b = 20 + a; }
   return 0;
}
```

This `while` loop is an example of an infinite loop. It is vital that at least one statement inside the `while` changes a variable in the test expression (in this example, the variable `a`); otherwise, the condition is always True. Because the variable `a` does not change inside the `while` loop, this program will never end.

TIP: If you inadvertently write an infinite loop, you must stop the program yourself. If you use a PC, this typically means pressing Ctrl-Break. If you are using a UNIX-based system, your system administrator might have to stop your program's execution.

3. The following program asks users for a first name, then uses a `while` loop to count the number of characters in the name. This is a *string length program;* it counts characters until it reaches the null zero. Remember that the length of a string equals the number of characters in the string, not including the null zero.

```
// Filename: C12WHIL2.CPP
// Counts the number of letters in the user's first name.
#include <iostream.h>
main()
{
   char name[15];           // Will hold user's first name
```

```
   int count=0;       // Will hold total characters in name

   // Get the user's first name
   cout << "What is your first name? ";
   cin >> name;

   while (name[count] > 0) // Loop until null zero reached.
      {  count++; }                  // Add 1 to the count.

   cout << "Your name has " << count << " characters";
   return 0;
}
```

The loop continues as long as the value of the next character in the `name` array is greater than zero. Because the last character in the array is a null zero, the test is False on the name's last character and the statement following the body of the loop continues.

NOTE: A built-in string function called `strlen()` determines the length of strings. You learn about this function in Chapter 22, "Character, String, and Numeric Functions."

4. The previous string-length program's `while` loop is not as efficient as it could be. Because a `while` loop fails when its test expression is zero, there is no need for the greater-than test. By changing the test expression as the following program shows, you can improve the efficiency of the string length count.

```
// Filename: C12WHIL3.CPP
// Counts the number of letters in the user's first name.
#include <iostream.h>
main()
{
   char name[15];           // Will hold user's first name
   int count=0;       // Will hold total characters in name

   // Get the user's first name
```

```
   cout << "What is your first name? ";
   cin >> name;

   while (name[count])  // Loop until null zero is reached.
      {  count++; }                  // Add 1 to the count.

   cout << "Your name has " << count << " characters";
   return 0;
}
```

The do-while Loop

The body of the do-while loop executes at least once.

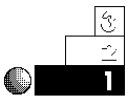
The do-while statement controls the do-while loop, which is similar to the while loop except the relational test occurs at the end (rather than beginning) of the loop. This ensures the body of the loop executes at least once. The do-while tests for a *positive relational test;* as long as the test is True, the body of the loop continues to execute.

The format of the do-while is

```
do
   { block of one or more C++ statements; }
while (test expression)
```

test expression must be enclosed in parentheses, just as it must in a while statement.

Examples



1. The following program is just like the first one you saw with the while loop (C12WHIL1.CPP), except the do-while is used. Notice the placement of test expression. Because this expression concludes the loop, user input does not have to appear before the loop and again in the body of the loop.

```
// Filename: C12WHIL4.CPP
// Input routine to ensure user types a
// correct response. This routine might be part
// of a larger program.
```

```
#include <iostream.h>
main()
{
   char ans;

   do
    { cout << "\nYou must type a Y or an N\n";    // Warn
                                                  // and ask
      cout << "Do you want to continue (Y/N) ?"; // again.
      cin >> ans; }            // Body of while loop
                               //    ends here.
   while ((ans != 'Y') && (ans != 'N'));

   return 0;
}
```

2. Suppose you are entering sales amounts into the computer to calculate extended totals. You want the computer to print the quantity sold, part number, and extended total (quantity times the price per unit), as the following program does.


```
// Filename: C12INV1.CPP
// Gets inventory information from user and prints
// an inventory detail listing with extended totals.
#include <iostream.h>
#include <iomanip.h>
main()
{
   int part_no, quantity;
   float cost, ext_cost;

   cout << "*** Inventory Computation ***\n\n";   // Title

   // Get inventory information.
   do
    { cout << "What is the next part number (-999 to end)? ";
      cin >> part_no;
      if (part_no != -999)
         { cout << "How many were bought? ";
           cin >> quantity;
           cout << "What is the unit price of this item? ";
```

```
            cin >> cost;
            ext_cost = cost * quantity;
            cout << "\n" << quantity << " of # " << part_no <<
                    " will cost " <<  setprecision(2) <<
                    ext_cost;
            cout << "\n\n\n";      // Print two blank lines.
           }
      } while (part_no != -999);       // Loop only if part
                                       //  number is not -999.

   cout << "End of inventory computation\n";
   return 0;
}
```

Here is the output from this program:

```
*** Inventory Computation ***

What is the next part number (-999 to end)? 213
How many were bought? 12
What is the unit price of this item? 5.66

12 of # 213 will cost 67.92


What is the next part number (-999 to end)? 92
How many were bought? 53
What is the unit price of this item? .23

53 of # 92 will cost 12.19


What is the next part number (-999 to end)? -999
End of inventory computation
```

The `do-while` loop controls the entry of the customer sales information. Notice the "trigger" that ends the loop. If the user enters –999 for the part number, the `do-while` loop quits because no part numbered –999 exists in the inventory.

However, this program can be improved in several ways. The invoice can be printed to the printer rather than the

screen. You learn how to direct your output to a printer in Chapter 21, “Device and Character Input/Output.” Also, the inventory total (the total amount of the entire order) can be computed. You learn how to total such data in the “Counters and Totals” section later in this chapter.

The `if` Loop Versus the `while` Loop

Some beginning programmers confuse the `if` statement with loop constructs. The `while` and `do-while` loops repeat a section of code multiple times, depending on the condition being tested. The `if` statement may or may not execute a section of code; if it does, it executes that section only once.

Use an `if` statement when you want to conditionally execute a section of code *once*, and use a `while` or `do-while` loop if you want to execute a section *more than once*. Figure 12.1 shows differences between the `if` statement and the two `while` loops.

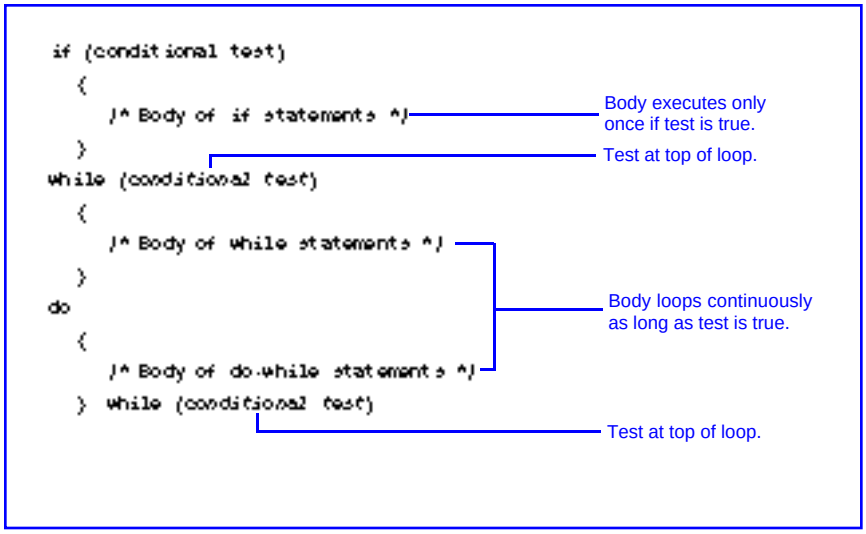


Figure 12.1. Differences between the `if` statement and the two `while` loops.

The exit() Function and break Statement

C++ provides the exit() function as a way to leave a program early (before its natural finish). The format of exit() is

```
exit(status);
```

where status is an optional integer variable or literal. If you are familiar with your operating system's return codes, status enables you to test the results of C++ programs. In DOS, status is sent to the operating system's errorlevel *environment variable*, where it can be tested by batch files.

The exit() function provides an early exit from your program.

Many times, something happens in a program that requires the program's termination. It might be a major problem, such as a disk drive error. Perhaps users indicate that they want to quit the program—you can tell this by giving your users a special value to type with cin or scanf(). You can isolate the exit() function on a line by itself, or anywhere else that a C++ statement or function can appear. Typically, exit() is placed in the body of an if statement to end the program early, depending on the result of some relational test.

Always include the stdlib.h header file when you use exit(). This file describes the operation of exit() to your program. Whenever you use a function in a program, you should know its corresponding #include header file, which is usually listed in the compiler's reference manual.

The break statement ends the current loop.

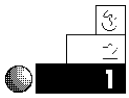
Instead of exiting an entire program, however, you can use the break statement to exit the current loop. The format of break is


```
break;
```

The break statement can go anywhere in a C++ program that any other statement can go, but it typically appears in the body of a while or do-while loop, used to leave the loop early. The following examples illustrate the exit() function and the break statement.

NOTE: The break statement exits only the most current loop. If you have a while loop in another while loop, break exits only the internal loop.

Examples



1. Here is a simple program that shows you how the `exit()` function works. This program looks as though it prints several messages on-screen, but it doesn't. Because `exit()` appears early in the code, this program quits immediately after `main()`'s opening brace.

```
// C12EXIT1.CPP
// Quits early due to exit() function.
#include <iostream.h>
#include <stdlib.h>              // Required for exit().
main()
{
   exit(0);                 // Forces program to end here.

   cout << "C++ programming is fun.\n";
   cout << "I like learning C++ by example!\n";
   cout << "C++ is a powerful language that is " <<
           "not difficult to learn.";

   return 0;
}
```

2. The `break` statement is not intended to be as strong a program exit as the `exit()` function. Whereas `exit()` ends the entire program, `break` quits only the loop that is currently active. In other words, `break` is usually placed inside a `while` or `do-while` loop to "simulate" a finished loop. The statement following the loop executes after a `break` occurs, but the program does not quit as it does with `exit()`.

   The following program appears to print `C++ is fun!` until the user enters `N` to stop it. The message prints only once, however, because the `break` statement forces an early exit from the loop.

```
// Filename: C12BRK.CPP
// Demonstrates the break statement.
#include <iostream.h>
main()
```

```
{
   char user_ans;

   do
     { cout << "C++ is fun! \n";
       break;                      // Causes early exit.
       cout << "Do you want to see the message again (N/Y)? ";
       cin >> user_ans;
     } while (user_ans == 'Y');

   cout << "That's all for now\n";
   return 0;
}
```

This program always produces the following output:

```
C++ is fun!
That's all for now
```

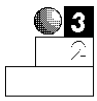
You can tell from this program's output that the `break` statement does not allow the `do-while` loop to reach its natural conclusion, but causes it to finish early. The final `cout` prints because only the current loop—and not the entire program—exits with the `break` statement.

3. Unlike the previous program, `break` usually appears after an `if` statement. This makes it a *conditional* `break`, which occurs only if the relational test of the `if` statement is True.

   A good illustration of this is the inventory program you saw earlier (C12INV1.CPP). Even though the users enter –999 when they want to quit the program, an additional `if` test is needed inside the `do-while`. The –999 ends the `do-while` loop, but the body of the `do-while` still needs an `if` test, so the remaining quantity and cost prompts are not given.

   If you insert a `break` after testing for the end of the user's input, as shown in the following program, the `do-while` will not need the `if` test. The `break` quits the `do-while` as soon as the user signals the end of the inventory by entering –999 as the part number.

```
// Filename: C12INV2.CPP
// Gets inventory information from user and prints
// an inventory detail listing with extended totals.
#include <iostream.h>
#include <iomanip.h>
main()
{
   int part_no, quantity;
   float cost, ext_cost;

   cout << "*** Inventory Computation ***\n\n";   // Title

   // Get inventory information
   do
    { cout << "What is the next part number (-999 to end)? ";
      cin >> part_no;
      if (part_no == -999)
        { break; }                  // Exit the loop if
                                    // no more part numbers.
      cout << "How many were bought? ";
      cin >> quantity;
      cout << "What is the unit price of this item? ";
      cin >> cost;
      cout << "\n" << quantity << " of # " << part_no <<
           " will cost " << setprecision(2) << cost*quantity;
      cout << "\n\n\n";           // Print two blank lines.
    } while (part_no != -999);        // Loop only if part
                                      // number is not -999.

   cout << "End of inventory computation\n";
   return 0;
}
```



4. You can use the following program to control the two other programs. This program illustrates how C++ can pass information to DOS with `exit()`. This is your first example of a menu program. Similar to a restaurant menu, a C++ menu program lists possible user choices. The users decide what they want the computer to do from the menu's available options. The mailing list application in Appendix F, "The Mailing List Application," uses a menu for its user options.

This program returns either a `1` or a `2` to its operating system, depending on the user's selection. It is then up to the operating system to test the `exit` value and run the proper program.

```
// Filename: C12EXIT2.CPP
// Asks user for his or her selection and returns
// that selection to the operating system with exit().
#include <iostream.h>
#include <stdlib.h>
main()
{
   int ans;

   do
     { cout << "Do you want to:\n\n";
       cout << "\t1. Run the word processor \n\n";
       cout << "\t2. Run the database program \n\n";
       cout << "What is your selection? ";
       cin >> ans;
     } while ((ans != 1) && (ans != 2));  // Ensures user
                                          // enters 1 or 2.
   exit(ans);  // Return value to operating system.
   return 0;   // Return does not ever execute due to exit().
}
```

Counters and Totals

Counting is important for many applications. You might have to know how many customers you have or how many people scored over a certain average in your class. You might want to count how many checks you wrote in the previous month with your computerized checkbook system.

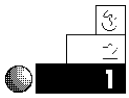
Before you develop C++ routines to count occurrences, think of how you count in your own mind. If you were adding a total number of something, such as the stamps in your stamp collection or the

number of wedding invitations you sent out, you would probably do the following:

Start at 0, and add 1 for each item being counted. When you are finished, you should have the total number (or the total count).

This is all you do when you count with C++: Assign 0 to a variable and add 1 to it every time you process another data value. The increment operator (++) is especially useful for counting.

Examples



1. To illustrate using a counter, the following program prints "Computers are fun!" on-screen 10 times. You can write a program that has 10 cout statements, but that would not be efficient. It would also be too cumbersome to have 5000 cout statements, if you wanted to print that same message 5000 times.

   By adding a while loop and a counter that stops after a certain total is reached, you can control this printing, as the following program shows.

```
// Filename: C12CNT1.CPP
// Program to print a message 10 times.
#include <iostream.h>
main()
{
   int ctr = 0;    // Holds the number of times printed.

   do
     { cout << "Computers are fun!\n";
       ctr++;                    // Add one to the count,
                                 // after each cout.
     } while (ctr < 10);         // Print again if fewer
                                 // than 10 times.
   return 0;
}
```

The output from this program is shown as follows. Notice that the message prints exactly 10 times.

```
Computers are fun!
Computers are fun!
Computers are fun!
Computers are fun!
Computers are fun!
Computers are fun!
Computers are fun!
Computers are fun!
Computers are fun!
Computers are fun!
```

The heart of the counting process in this program is the statement that follows.

```
ctr++;
```

You learned earlier that the increment operator adds 1 to a variable. In this program, the counter variable is incremented each time the `do-while` loops. Because the only operation performed on this line is the increment of `ctr`, the prefix increment (`++ctr`) produces the same results.

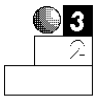
2. The previous program not only added to the counter variable, but also performed the loop a specific number of times. This is a common method of conditionally executing parts of a program for a fixed number of times.

   The following program is a password program. A password is stored in an integer variable. The user must correctly enter the matching password in three attempts. If the user does not type the correct password in that time, the program ends. This is a common method that dial-up computers use. They enable a caller to try the password a fixed number of times, then hang up the phone if that limit is exceeded. This helps deter people from trying hundreds of different passwords at any one sitting.

   If users guess the correct password in three tries, they see the secret message.

```
// Filename: C12PASS1.CPP
// Program to prompt for a password and
// check it against an internal one.
#include <iostream.h>
#include <stdlib.h>
main()
{
   int stored_pass = 11862;
   int num_tries = 0;     // Counter for password attempts.
   int user_pass;

   while (num_tries < 3)              // Loop only three
                                      // times.
      { cout << "What is the password (You get 3 tries...)? ";
        cin >> user_pass;
        num_tries++;                   // Add 1 to counter.
        if (user_pass == stored_pass)
           { cout << "You entered the correct password.\n";
             cout << "The cash safe is behind the picture " <<
                     "of the ship.\n";
             exit(0);
           }
        else
           { cout << "You entered the wrong password.\n";
          if (num_tries == 3)
          { cout << "Sorry, you get no more chances"; }
          else
          { cout << "You get " << (3-num_tries) <<
                  " more tries...\n";}
        }
      }                               // End of while loop.
   exit(0);
   return 0;
}
```

This program gives users three chances in case they type some mistakes. After three unsuccessful attempts, the program quits without displaying the secret message.



3. The following program is a letter-guessing game. It includes a message telling users how many tries they made before guessing the correct letter. A counter counts the number of these tries.

```
// Filename: C12GUES.CPP
// Letter-guessing game.
#include <iostream.h>
main()
{
   int tries = 0;
   char comp_ans, user_guess;

   // Save the computer's letter
   comp_ans = 'T';                // Change to a different
                                  //     letter if desired.

   cout << "I am thinking of a letter...";
   do
      { cout << "What is your guess? ";
        cin >> user_guess;
        tries++;   // Add 1 to the guess-counting variable.
        if (user_guess > comp_ans)
           { cout << "Your guess was too high\n";
             cout << "\nTry again...";
           }
        if (user_guess < comp_ans)
           { cout << "Your guess was too low\n";
             cout << "\nTry again...";
           }
       } while (user_guess != comp_ans);  // Quit when a
                                          // match is found.

   // They got it right, let them know.
   cout << "*** Congratulations!  You got it right! \n";
   cout << "It took you only " << tries <<
           " tries to guess.";
   return 0;
}
```

Here is the output of this program:

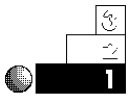
```
I am thinking of a letter...What is your guess? E
Your guess was too low

Try again...What is your guess? X
Your guess was too high

Try again...What is your guess? H
Your guess was too low

Try again...What is your guess? O
Your guess was too low

Try again...What is your guess? U
Your guess was too high

Try again...What is your guess? Y
Your guess was too high

Try again...What is your guess? T
*** Congratulations!  You got it right!
It took you only 7 tries to guess.
```

Producing Totals

Writing a routine to add values is as easy as counting. Instead of adding 1 to the counter variable, you add a value to the total variable. For instance, if you want to find the total dollar amount of checks you wrote during December, you can start at nothing (0) and add the amount of every check written in December. Instead of building a count, you are building a total.

When you want C++ to add values, just initialize a `total` variable to zero, then add each value to the total until you have included all the values.

Examples



1. Suppose you want to write a program that adds your grades for a class you are taking. The teacher has informed you that you earn an *A* if you can accumulate over 450 points.

   The following program keeps asking you for values until you type –1. The –1 is a signal that you are finished entering grades and now want to see the total. This program also prints a congratulatory message if you have enough points for an *A*.

```
// Filename: C12GRAD1.CPP
// Adds grades and determines whether you earned an A.
#include <iostream.h>
include <iomanip.h>
main()
{
   float total_grade=0.0;
   float grade;              // Holds individual grades.

   do
   { cout << "What is your grade? (-1 to end) ";
     cin >> grade;
     if (grade >= 0.0)
      { total_grade += grade; }        // Add to total.
   } while (grade >= 0.0);     // Quit when -1 entered.

   // Control begins here if no more grades.
   cout << "\n\nYou made a total of " << setprecision(1) <<
            total_grade << " points\n";
   if (total_grade >= 450.00)
      { cout << "** You made an A!!"; }

   return 0;
}
```

Notice that the `-1` response is not added to the total number of points. This program checks for the `-1` before adding to `total_grade`. Here is the output from this program:

```
What is your grade? (-1 to end) 87.6
What is your grade? (-1 to end) 92.4
What is your grade? (-1 to end) 78.7
What is your grade? (-1 to end) -1

You made a total of  258.7 points
```

2. The following program is an extension of the grade-calculating program. It not only totals the points, but also computes their average.

   To calculate the average grade, the program must first determine how many grades were entered. This is a subtle problem because the number of grades to be entered is unknown in advance. Therefore, every time the user enters a valid grade (not –1), the program must add 1 to a counter as well as add that grade to the `total` variable. This is a combination counting and totaling routine, which is common in many programs.

```
// Filename: C12GRAD2.CPP
// Adds up grades, computes average,
// and determines whether you earned an A.
#include <iostream.h>
#include <iomanip.h>
main()
{
   float total_grade=0.0;
   float grade_avg = 0.0;
   float grade;
   int grade_ctr = 0;

   do
   { cout << "What is your grade? (-1 to end) ";
     cin >> grade;
     if (grade >= 0.0)
      { total_grade += grade;              // Add to total.
        grade_ctr ++; }                    // Add to count.
   } while (grade >= 0.0);          // Quit when -1 entered.
```

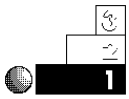
```
    // Control begins here if no more grades.
    grade_avg = (total_grade / grade_ctr);       // Compute
                                                 // average.
    cout << "\nYou made a total of " << setprecision(1) <<
            total_grade << " points.\n";
    cout << "Your average was " << grade_avg << "\n";
    if (total_grade >= 450.0)
       { cout << "** You made an A!!"; }
    return 0;
}
```

Below is the output of this program. Congratulations! You are on your way to becoming a master C++ programmer.

```
What is your grade? (-1 to end) 67.8
What is your grade? (-1 to end) 98.7
What is your grade? (-1 to end) 67.8
What is your grade? (-1 to end) 92.4
What is your grade? (-1 to end) -1

You made a total of 326.68 points.
Your average was 81.7
```

Review Questions

The answers to the review questions are in Appendix B.



1. What is the difference between the `while` loop and the `do-while` loop?
2. What is the difference between a total variable and a counter variable?
3. Which C++ operator is most useful for counting?
4. True or false: Braces are not required around the body of `while` and `do-while` loops.

5. What is wrong with the following code?

```
while (sales > 50)
   cout << "Your sales are very good this month.\n";
   cout << "You will get a bonus for your high sales\n";
```

6. What file must you include as a header file if you use `exit()`?

7. How many times does this `printf()` print?

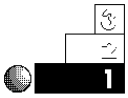
```
int a=0;
do
   { printf("Careful \n");
     a++; }
while (a > 5);
```

8. How can you inform DOS of the program exit status?

9. What is printed to the screen in the following section of code?

```
a = 1;
while (a < 4)
   { cout << "This is the outer loop\n";
     a++;
     while (a <= 25)
        { break;
          cout << "This prints 25 times\n"; }
   }
```

Review Exercises



1. Write a program with a `do-while` loop that prints the numerals from 10 to 20 (inclusive), with a blank line between each number.

2. Write a weather-calculator program that asks for a list of the previous 10 days' temperatures, computes the average, and prints the results. You have to compute the total as the input occurs, then divide that total by 10 to find the average. Use a `while` loop for the 10 repetitions.

3. Rewrite the program in Exercise 2 using a `do-while` loop.

4. Write a program, similar to the weather calculator in Exercise 2, but generalize it so it computes the average of any number of days' temperatures. (*Hint:* You have to count the number of temperatures to compute the final average.)

5. Write a program that produces your own ASCII table on-screen. Don't print the first 31 characters because they are nonprintable. Print the codes numbered 32 through 255 by storing their numbers in integer variables and printing their ASCII values using `printf()` and the `"%c"` format code.

Summary

This chapter showed you two ways to produce a C++ loop: the `while` loop and the `do-while` loop. These two variations of `while` loops differ in where they test their `test condition` statements. The `while` tests at the beginning of its loop, and the `do-while` tests at the end. Therefore, the body of a `do-while` loop always executes at least once. You also learned that the `exit()` function and `break` statement add flexibility to the `while` loops. The `exit()` function terminates the program, and the `break` statement terminates only the current loop.

This chapter explained two of the most important applications of loops: counters and totals. Your computer can be a wonderful tool for adding and counting, due to the repetitive capabilities offered with `while` loops.

The next chapter extends your knowledge of loops by showing you how to create a *determinate* loop, called the `for` loop. This feature is useful when you want a section of code to loop for a specified number of times.

Arrays of Structures

This chapter builds on the previous one by showing you how to create many structures for your data. After creating an array of structures, you can store many occurrences of your data values.

Arrays of structures are good for storing a complete employee file, inventory file, or any other set of data that fits in the structure format. Whereas arrays provide a handy way to store several values that are the same type, arrays of structures store several values of different types together, grouped as structures.

This chapter introduces the following concepts:

- Creating arrays of structures
- Initializing arrays of structures
- Referencing elements from a structure array
- Arrays as members

Many C++ programmers use arrays of structures as a prelude to storing their data in a disk file. You can input and calculate your disk data in arrays of structures, and then store those structures in memory. Arrays of structures also provide a means of holding data you read from the disk.

Declaring Arrays of Structures

It is easy to declare an array of structures. Specify the number of reserved structures inside array brackets when you declare the structure variable. Consider the following structure definition:

```
struct stores
     { int employees;
          int registers;
          double sales;
     } store1, store2, store3, store4, store5;
```

This structure should not be difficult for you to understand because there are no new commands used in the structure declaration. This structure declaration creates five structure variables. Figure 29.1 shows how C++ stores these five structures in memory. Each of the structure variables has three members—two integers followed by a double floating-point value.

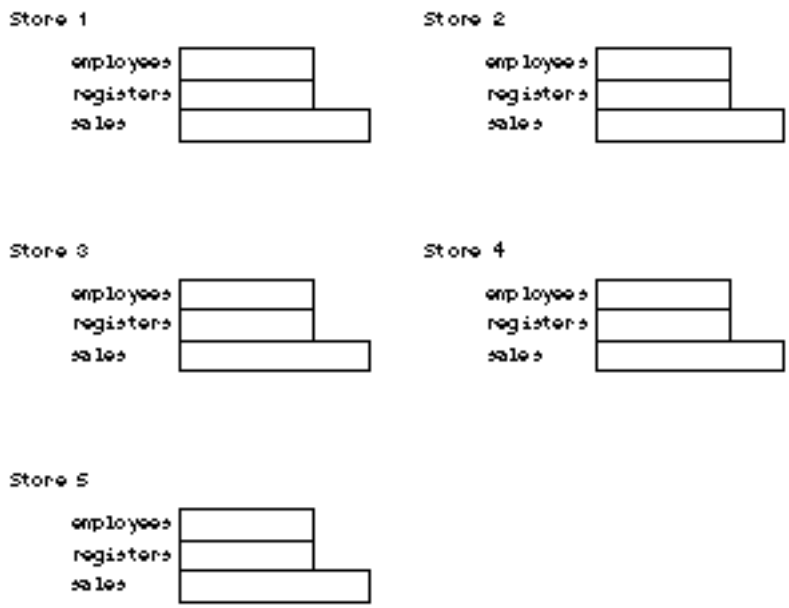


Figure 29.1. The structure of `Store 1`, `Store 2`, `Store 3`, `Store 4`, and `Store 5`.

If the fourth store increased its employee count by three, you could update the store's employee number with the following assignment statement:

```
store4.employees += 3;          // Add three to this store's
                                // employee count.
```

Suppose the fifth store just opened and you want to initialize its members with data. If the stores are a chain and the new store is similar to one of the others, you can begin initializing the store's data by assigning each of its members the same data as another store's, like this:

```
store5 = store2;                // Define initial values for
                                // the members of store5.
```

Arrays of structures make working with large numbers of structure variables manageable.

Such structure declarations are fine for a small number of structures, but if the stores were a national chain, five structure variables would not be enough. Suppose there were 1000 stores. You would not want to create 1000 different store variables and work with each one individually. It would be much easier to create an array of store structures.

Consider the following structure declaration:

```
struct stores
     { int employees;
          int registers;
          double sales;
     } store[1000];
```

In one quick declaration, this code creates 1000 store structures, each one containing three members. Figure 29.2 shows how these structure variables appear in memory. Notice the name of each individual structure variable: `store[0]`, `store[1]`, `store[2]`, and so on.

CAUTION: Be careful that your computer does not run out of memory when you create a large number of structures. Arrays of structures quickly consume valuable memory. You might have to create fewer structures, storing more data in disk files and less data in memory.

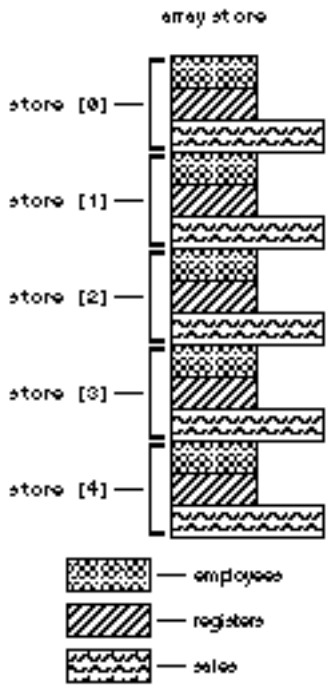


Figure 29.2. An array of the `store` structures.

The element `store[2]` is an array element. This array element, unlike the others you have seen, is a structure variable. Therefore, it contains three members, each of which you can reference with the dot operator.

The dot operator works the same way for structure array elements as it does for regular structure variables. If the number of employees for the fifth store (`store[4]`) increased by three, you could update the structure variable like this:

```
store[4].employees += 3;       // Add three to this store's
                               // employee count.
```

You can assign complete structures to one another also by using array notation. To assign all the members of the 20th store to the 45th store, you would do this:

```
store[44] = store[19];     // Copy all members from the
                           // 20th store to the 45th.
```

The rules of arrays are still in force here. Each element of the array called `store` is the same data type. The data type of `store` is the structure `stores`. As with any array, each element must be the same data type; you cannot mix data types in the same array. This array's data type happens to be a structure you created containing three members. The data type for `store[316]` is the same for `store[981]` and `store[74]`.

The name of the array, `store`, is a pointer constant to the starting element of the array, `store[0]`. Therefore, you can use pointer notation to reference the stores. To assign `store[60]` the same value as `store[23]`, you can reference the two elements like this:

```
*(store+60) = *(store+23);
```

You also can mix array and pointer notation, such as

```
store[60] = *(store+23);
```

and receive the same results.

You can increase the sales of `store[8]` by 40 percent using pointer or subscript notation as well, as in

```
store[8].sales = (*(store+8)).sales * 1.40;
```

The extra pair of parentheses are required because the dot operator has precedence over the dereferencing symbol in C++'s hierarchy of operators (see Appendix D, "C++ Precedence Table"). Of course, in this case, the code is not helped by the pointer notation. The following is a much clearer way to increase the `sales` by 40 percent:

```
store[8].sales *= 1.40;
```

The following examples build an inventory data-entry system for a mail-order firm using an array of structures. There is very little new you have to know when working with arrays of structures. To become comfortable with the arrays of structure notation, concentrate on the notation used when accessing arrays of structures and their members.

Keep Your Array Notation Straight

You would never access the member `sales` like this:

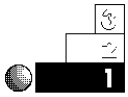
```
store.sales[8] = 3234.54;          // Invalid
```

Array subscripts follow only array elements. `sales` is not an array; it was declared as being a double floating-point number. `store` can never be used without a subscript (unless you are using pointer notation).

Here is a corrected version of the previous assignment statement:

```
store[8].sales=3234.54;            // Correctly assigns
                                   // the value.
```

Examples



1. Suppose you work for a mail-order company that sells disk drives. You are given the task of writing a tracking program for the 125 different drives you sell. You must keep track of the following information:

   Storage capacity in megabytes
   Access time in milliseconds
   Vendor code (A, B, C, or D)
   Cost
   Price

   Because there are 125 different disk drives in the inventory, the data fits nicely into an array of structures. Each array element is a structure containing the five members described in the list.

   The following structure definition defines the inventory:

```
struct inventory
{
```

```
    long int storage;
    int access_time;
    char vendor_code;
    double code;
    double price;
} drive[125];  // Defines 125 occurrences of the structure.
```

2. When working with a large array of structures, your first concern should be how the data inputs into the array elements. The best method of data-entry depends on the application.

   For example, if you are converting from an older computerized inventory system, you have to write a conversion program that reads the inventory file in its native format and saves it to a new file in the format required by your C++ programs. This is no easy task. It demands that you have extensive knowledge of the system from which you are converting.

   If you are writing a computerized inventory system for the first time, your job is a little easier because you do not have to convert the old files. You still must realize that someone has to type the data into the computer. You must write a data-entry program that receives each inventory item from the keyboard and saves it to a disk file. You should give the user a chance to edit inventory data to correct any data he or she originally might have typed incorrectly.

   One of the reasons disk files are introduced in the last half of the book is that disk-file formats and structures share a common bond. When you store data in a structure, or more often, in an array of structures, you can easily write that data to a disk file using straightforward disk I/O commands.

   The following program takes the array of disk drive structures shown in the previous example and adds a data-entry function so the user can enter data into the array of structures. The program is menu-driven. The user has a choice, when starting the program, to add data, print data on-screen, or exit the program. Because you have yet to see disk I/O commands, the data in the array of structures goes away

when the program ends. As mentioned earlier, saving those structures to disk is an easy task after you learn C++'s disk I/O commands. For now, concentrate on the manipulation of the structures.

This program is longer than many you previously have seen in this book, but if you have followed the discussions of structures and the dot operator, you should have little trouble following the code.

Identify the program and include the necessary header files. Define a structure that describes the format of each inventory item. Create an array of structures called `disk`.

Display a menu that gives the user the choice of entering new inventory data, displaying the data on-screen, or quitting the program. If the user wants to enter new inventory items, prompt the user for each item and store the data into the array of structures. If the user wants to see the inventory, loop through each inventory item in the array, displaying each one on-screen.

```
// Filename: C29DSINV.CPP
// Data-entry program for a disk drive company.
#include <iostream.h>
#include <stdlib.h>
#include <iomanip.h>
#include <stdio.h>

struct inventory            // Global structure definition.
{
   long int storage;
   int access_time;
   char vendor_code;
   float cost;
   float price;
};              // No structure variables defined globally.

void disp_menu(void);
struct inventory enter_data();
void see_data(inventory disk[125], int num_items);

void main()
```

```
{
   inventory disk[125];   // Local array of structures.
   int ans;
   int num_items=0;               // Number of total items
                                  // in the inventory.

   do
     {
        do
         { disp_menu();    // Display menu of user choices.
           cin >> ans;              // Get user's request.
         } while ((ans<1) || (ans>3));

         switch (ans)
      { case (1): { disk[num_items] = enter_data(); // Enter
                                                    // disk data.
                num_items++;   // Increment number of items.
                break; }
        case (2): { see_data(disk, num_items);  // Display
                                                    // disk data.
                break; }
        default : { break; }
      }
      }  while (ans!=3);           // Quit program
                                   // when user is done.
     return;
}

void disp_menu(void)
{

   cout << "\n\n*** Disk Drive Inventory System ***\n\n";
   cout << "Do you want to:\n\n";
   cout << "\t1. Enter new item in inventory\n\n";
   cout << "\t2. See inventory data\n\n";
   cout << "\t3. Exit the program\n\n";
   cout << "What is your choice? ";
   return;
}

inventory enter_data()
```

```
{
   inventory disk_item;    // Local variable to fill
                           // with input.

   cout << "\n\nWhat is the next drive's storage in bytes? ";
   cin >> disk_item.storage;
   cout << "What is the drive's access time in ms? ";
   cin >> disk_item.access_time;
   cout << "What is the drive's vendor code (A, B, C, or D)? ";
   fflush(stdin);  // Discard input buffer
                   // before accepting character.
   disk_item.vendor_code = getchar();
   getchar();  // Discard carriage return
   cout << "What is the drive's cost? ";
   cin >> disk_item.cost;
   cout << "What is the drive's price? ";
   cin >> disk_item.price;

   return (disk_item);
}

void see_data(inventory disk[125], int num_items)
{
   int ctr;
   cout << "\n\nHere is the inventory listing:\n\n";
   for (ctr=0;ctr<num_items;ctr++)
      {
      cout << "Storage: " << disk[ctr].storage << "\t";
      cout << "Access time: " << disk[ctr].access_time << "\n";
      cout << "Vendor code: " << disk[ctr].vendor_code << "\t";
      cout << setprecision(2);
      cout << "Cost: $" << disk[ctr].cost << "\t";
      cout << "Price: $" << disk[ctr].price << "\n";
      }
   return;

}
```

Figure 29.3 shows an item being entered into the inventory file. Figure 29.4 shows the inventory listing being displayed to the screen. There are many features and error-checking functions you can add, but this program is the foundation of a more comprehensive inventory system. You can easily

adapt it to a different type of inventory, a video tape collection, a coin collection, or any other tracking system by changing the structure definition and the member names throughout the program.

```
*** Disk Drive Inventory System ***

Do you want to:

        1. Enter new item in inventory

        2. See inventory data

        3. Exit the program

What is your choice? 1


What is the next drive's storage in bytes? 120000
What is the drive's access time in ms? 17
What is the drive's vendor code (A, B, C, or D)? A
What is the drive's cost? 121.56
What is the drive's price? 240.00
```

Figure 29.3. Entering inventory information.

Arrays as Members

Members of structures can be arrays. Array members pose no new problems, but you have to be careful when you access individual array elements. Keeping track of arrays of structures that contain array members might seem like a great deal of work on your part, but there is nothing to it.

Consider the following structure definition. This statement declares an array of 100 structures, each structure holding payroll information for a company. Two of the members, `name` and `department`, are arrays.

```
struct payroll
   { char name[25];                  // Employee name array.
```

```
    int dependents;
    char department[10];        // Department name array.
    float salary;
} employee[100];           // An array of 100 employees.
```

```
What is your choice? 2

Here is the inventory listing:

Storage: 120000 Access time: 17
Vendor code: A  Cost: $121.56   Price: $240.00
Storage: 320000 Access time: 21
Vendor code: D  Cost: $230.85   Price: $409.57
Storage: 280000 Access time: 19
Vendor code: C  Cost: $210.84   Price: $398.67

*** Disk Drive Inventory System ***

Do you want to:

        1. Enter new item in inventory

        2. See inventory data

        3. Exit the program

What is your choice? 3
```

Figure 29.4. Displaying the inventory data.

Figure 29.5 shows what these structures look like. The first and third members are arrays. `name` is an array of 25 characters, and `department` is an array of 10 characters.

Suppose you must save the 25th employee's initial in a character variable. Assuming initial is already declared as a character variable, the following statement assigns the employee's initial to the varible `initial`:

```
initial = employee[24].name[0];
```

The double subscripts might look confusing, but the dot operator requires a structure variable on its left (`employee[24]`) and a member on its right (`name`'s first array element). Being able to refer to member arrays makes the processing of character data in structures simple.

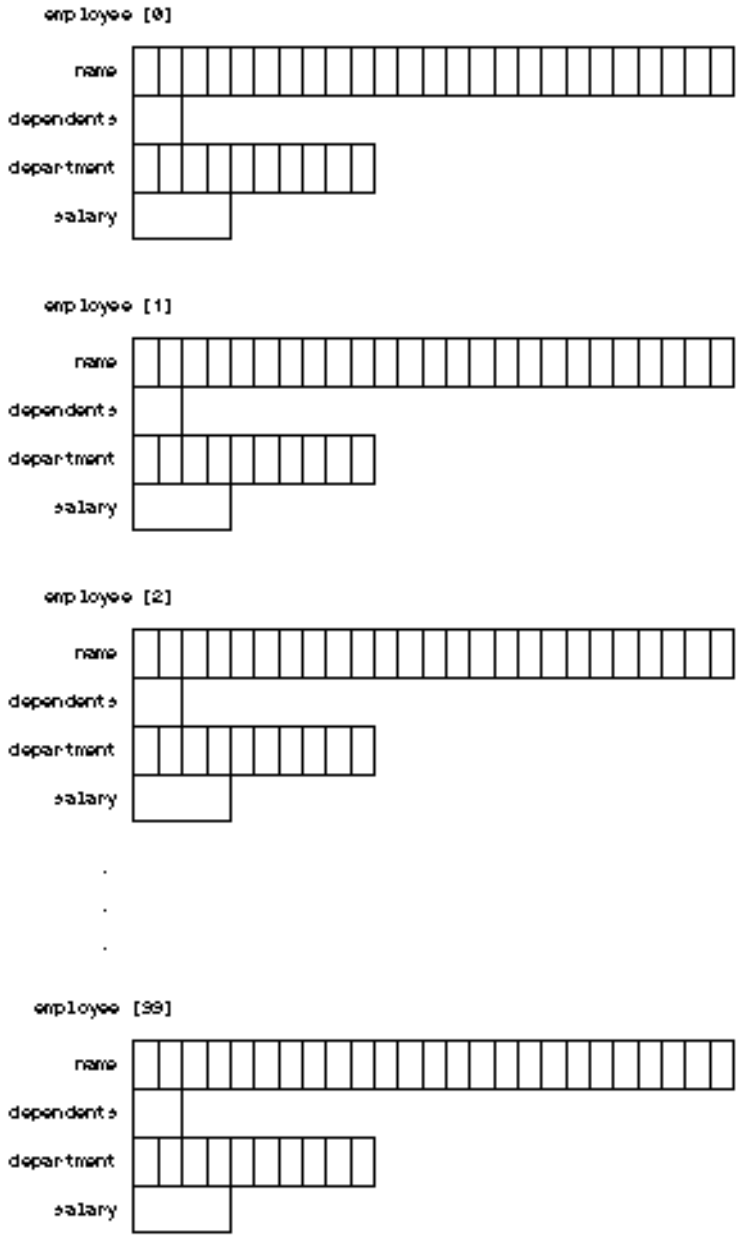
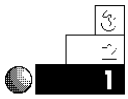


Figure 29.5. The payroll data.

Examples



1. Suppose an employee got married and wanted her name changed in the payroll file. (She happens to be the 45th employee in the array of structures.) Given the payroll structure described in the previous section, this would assign a new name to her structure:

```
strcpy(employee[44].name, "Mary Larson");    // Assign
                                             // a new name.
```

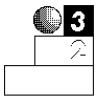
When you refer to a structure variable using the dot operator, you can use regular commands and functions to process the data in the structure members.

2. A bookstore wants to catalog its inventory of books. The following program creates an array of 100 structures. Each structure contains several types of variables, including arrays. This program is the data-entry portion of a larger inventory system. Study the references to the members to see how member-arrays are used.

```
// Filename: C29BOOK.CPP
// Bookstore data-entry program.
#include <iostream.h>
#include <stdio.h>
#include <ctype.h>

struct inventory
  { char title[25];                  // Book's title.
    char pub_date[19];               // Publication date.
    char author[20];                 // Author's name.
    int num;                         // Number in stock.
    int on_order;                    // Number on order.
    float retail;                    // Retail price.
  };

void main()
{
   inventory book[100];
   int total=0;                // Total books in inventory.
   int ans;
```

```
    do     // This program enters data into the structures.
       { cout << "Book #" << (total+1) << ":\n", (total+1);
      cout << "What is the title? ";
      gets(book[total].title);
      cout << "What is the publication date? ";
      gets(book[total].pub_date);
      cout << "Who is the author? ";
      gets(book[total].author);
      cout << "How many books of this title are there? ";
      cin >> book[total].num;
      cout << "How many are on order? ";
      cin >> book[total].on_order;
      cout << "What is the retail price? ";
      cin >> book[total].retail;
      fflush(stdin);
      cout << "\nAre there more books? (Y/N) ";
      ans=getchar();
      fflush(stdin);            // Discard carriage return.
      ans=toupper(ans);         // Convert to uppercase.
      if (ans=='Y')
        { total++;
          continue; }
       } while (ans=='Y');
    return;
}
```

You need much more to make this a usable inventory program. An exercise at the end of this chapter recommends ways you can improve on this program by adding a printing routine and a title and author search. One of the first things you should do is put the data-entry routine in a separate function to make the code more modular. Because this example is so short, and because the program performs only one task (data-entry), there was no advantage to putting the data-entry task in a separate function.



3. Here is a comprehensive example of the steps you might go through to write a C++ program. You should begin to understand the C++ language enough to start writing some advanced programs.

Assume you have been hired by a local bookstore to write a magazine inventory system. You have to track the following:

Magazine title (at most, 25 characters)
Publisher (at most, 20 characters)
Month (1, 2, 3,...12)
Publication year
Number of copies in stock
Number of copies on order
Price of magazine (dollars and cents)

Suppose there is a projected maximum of 1000 magazine titles the store will ever carry. This means you need 1000 occurrences of the structure, not 1000 magazines total. Here is a good structure definition for such an inventory:

```
struct mag_info
   { char title[25];
     char pub[25];
     int month;
     int year;
     int stock_copies;
     int order_copies;
     float price;
   } mags[1000];              // Define 1000 occurrences.
```

Because this program consists of more than one function, it is best to declare the structure globally, and the structure variables locally in the functions that need them.

This program needs three basic functions: a `main()` controlling function, a data-entry function, and a data printing function. You can add much more, but this is a good start for an inventory system. To keep the length of this example reasonable, assume the user wants to enter several magazines, then print them. (To make the program more "usable," you should add a menu so the user can control when she or he adds and prints the information, and should add more error-checking and editing capabilities.)

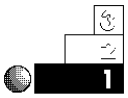
Here is an example of the complete data-entry and printing program with prototypes. The arrays of structures are passed between the functions from `main()`.


```
// Filename: C29MAG.CPP
// Magazine inventory program for adding and displaying
// a bookstore's magazines.
#include <iostream.h>
#include <ctype.h>
#include <stdio.h>

struct mag_info
   { char title[25];
     char pub[25];
     int month;
     int year;
     int stock_copies;
     int order_copies;
     float price;
   };

mag_info fill_mags(struct mag_info mag);
void print_mags(struct mag_info mags[], int mag_ctr);

void main()
{
   mag_info mags[1000];
   int mag_ctr=0;            // Number of magazine titles.
   char ans;

   do
   {                         // Assumes there is
                             // at least one magazine filled.
      mags[mag_ctr] = fill_mags(mags[mag_ctr]);
      cout << "Do you want to enter another magazine? ";
      fflush(stdin);
      ans = getchar();
      fflush(stdin);           // Discards carriage return.
      if (toupper(ans) == 'Y')
        { mag_ctr++; }
        } while (toupper(ans) == 'Y');
 print_mags(mags, mag_ctr);
```

```
      return;                // Returns to operating system.
}

void print_mags(mag_info mags[], int mag_ctr)
{
   int i;
   for (i=0; i<=mag_ctr; i++)
     { cout << "\n\nMagazine " << i+1 << ":\n";// Adjusts for
                                                // subscript.
       cout << "\nTitle: " << mags[i].title << "\n";
       cout << "\tPublisher: " << mags[i].pub << "\n";
       cout << "\tPub. Month: " << mags[i].month << "\n";
       cout << "\tPub. Year: " << mags[i].year << "\n";
       cout << "\tIn-stock: " << mags[i].stock_copies << "\n";
       cout << "\tOn order: " << mags[i].order_copies << "\n";
       cout << "\tPrice: " << mags[i].price << "\n";
     }
   return;
}

mag_info fill_mags(mag_info mag)
{
   puts("\n\nWhat is the title? ");
   gets(mag.title);
   puts("Who is the publisher? ");
   gets(mag.pub);
   puts("What is the month (1, 2, ..., 12)? ");
   cin >> mag.month;
   puts("What is the year? ");
   cin >> mag.year;
   puts("How many copies in stock? ");
   cin >> mag.stock_copies;
   puts("How many copies on order? ");
   cin >> mag.order_copies;
   puts("How much is the magazine? ");
   cin >> mag.price;
   return (mag);
}
```

Review Questions

The answers to the review questions are in Appendix B.



1. True or false: Each element in an array of structures must be the same type.
2. What is the advantage of creating an array of structures rather than using individual variable names for each structure variable?

3. Given the following structure declaration:

```
struct item
  { char part_no[8];
    char descr[20];
    float price;
    int in_stock;
  } inventory[100];
```

   a. How would you assign a price of 12.33 to the 33rd item's in-stock quantity?

   b. How would you assign the first character of the 12th item's part number the value of *X*?

   c. How would you assign the 97th inventory item the same values as the 63rd?

4. Given the following structure declaration:

```
struct item
  { char desc[20];
    int num;
    float cost;
  } inventory[25];
```

   a. What is wrong with the following statement?

```
item[1].cost = 92.32;
```

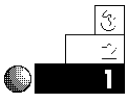
   b. What is wrong with the following statement?

```
strcpy(inventory.desc, "Widgets");
```

c. What is wrong with the following statement?

```
inventory.cost[10] = 32.12;
```

Review Exercises



1. Write a program that stores an array of friends' names, phone numbers, and addresses and prints them two ways: with their name, address, and phone number, or with only their name and phone number for a phone listing.

2. Add a sort function to the program in Exercise 1 so you can print your friends' names in alphabetical order. (*Hint:* You have to make the member holding the names a character pointer.)

3. Expand on the book data-entry program, C29BOOK.CPP, by adding features to make it more usable (such as search book by author, by title, and print an inventory of books on order).

Summary

You have mastered structures and arrays of structures. Many useful inventory and tracking programs can be written using structures. By being able to create arrays of structures, you can now create several occurrences of data.

The next step in the process of learning C++ is to save these structures and other data to disk files. The next two chapters explore the concepts of disk file processing.

Sequential Files

So far, every example in this book has processed data that resided inside the program listing or came from the keyboard. You assigned constants and variables to other variables and created new data values from expressions. The programs also received input with `cin`, `gets()`, and the character input functions.

The data created by the user and assigned to variables with assignment statements is sufficient for some applications. With the large volumes of data most real-world applications must process, however, you need a better way of storing that data. For all but the smallest computer programs, disk files offer the solution.

After storing data on the disk, the computer helps you enter, find, change, and delete the data. The computer and C++ are simply tools to help you manage and process data. This chapter focuses on disk- and file-processing concepts and teaches you the first of two methods of disk access, *sequential file access.*

This chapter introduces you to the following concepts:

- An overview of disk files
- The types of files
- Processing data on the disk
- Sequential file access
- File I/O functions

After this chapter, you will be ready to tackle the more advanced random-file-access methods covered in the next chapter. If you have programmed computerized data files with another programming language, you might be surprised at how C++ borrows from other programming languages, especially BASIC, when working with disk files. If you are new to disk-file processing, disk files are simple to create and to read.

Why Use a Disk?

The typical computer system has much less memory storage than hard disk storage. Your disk drive holds much more data than can fit in your computer's RAM. This is the primary reason for using the disk for storing your data. The disk memory, because it is nonvolatile, also lasts longer; when you turn your computer off, the disk memory is not erased, whereas RAM is erased. Also, when your data changes, you (or more important, your users) do not have to edit the program and look for a set of assignment statements. Instead, the users run previously written programs that make changes to the disk data.

Disks hold more data than computer memory.

This makes programming more difficult at first because programs have to be written to change the data on the disk. Nonprogrammers, however, can then use the programs and modify the data without knowing C++.

The capacity of your disk makes it a perfect place to store your data as well as your programs. Think about what would happen if all data had to be stored with a program's assignment statements. What if the Social Security Office in Washington, D.C., asked you to write a C++ program to compute, average, filter, sort, and print each person's name and address in his or her files? Would you want your program to include millions of assignment statements? Not only would you not want the program to hold that much data, but it could not do so because only relatively small amounts of data fit in a program before you run out of RAM.

By storing data on your disk, you are much less limited because you have more storage. Your disk can hold as much data as you have disk capacity. Also, if your program requirements grow, you can usually increase your disk space, whereas you cannot always add more RAM to your computer.

NOTE: C++ cannot access the special extended or expanded memory some computers have.

When working with disk files, C++ does not have to access much RAM because C++ reads data from your disk drive and processes the data only parts at a time. Not all your disk data has to reside in RAM for C++ to process it. C++ reads some data, processes it, and then reads some more. If C++ requires disk data a second time, it rereads that place on the disk.

Types of Disk File Access

Your programs can access files two ways: through sequential access or random access. Your application determines the method you should choose. The access mode of a file determines how you read, write, change, and delete data from the file. Some of your files can be accessed in both ways, sequentially and randomly as long as your programs are written properly and the data lends itself to both types of file access.

A sequential file has to be accessed in the same order the file was written. This is analogous to cassette tapes: You play music in the same order it was recorded. (You can quickly fast-forward or rewind over songs you do not want to listen to, but the order of the songs dictates what you do to play the song you want.) It is difficult, and sometimes impossible, to insert data in the middle of a sequential file. How easy is it to insert a new song in the middle of two other songs on a tape? The only way to truly add or delete records from the middle of a sequential file is to create a completely new file that combines both old and new records.

It might seem that sequential files are limiting, but it turns out that many applications lend themselves to sequential-file processing.

Unlike sequential files, you can access random-access files in any order you want. Think of data in a random-access file as you would songs on a compact disc or record; you can go directly to any song you want without having to play or fast-forward over the other songs. If you want to play the first song, the sixth song, and then the fourth song, you can do so. The order of play has nothing to do with the order in which the songs were originally recorded. Random-file access sometimes takes more programming but rewards your effort with a more flexible file-access method. Chapter 31 discusses how to program for random-access files.

Sequential File Concepts

There are three operations you can perform on sequential disk files. You can

- Create disk files
- Add to disk files
- Read from disk files

Your application determines what you must do. If you are creating a disk file for the first time, you must create the file and write the initial data to it. Suppose you wanted to create a customer data file. You would create a new file and write your current customers to that file. The customer data might originally be in arrays, arrays of structures, pointed to with pointers, or placed in regular variables by the user.

Over time, as your customer base grows, you can add new customers to the file (called *appending* to the file). When you add to the end of a file, you append to that file. As your customers enter your store, you would read their information from the customer data file.

Customer disk processing is an example of one disadvantage of sequential files, however. Suppose a customer moves and wants you to change his or her address in your files. Sequential-access files do not lend themselves well to changing data stored in them. It is also difficult to remove information from sequential files. Random files, described in the next chapter, provide a much easier approach

to changing and removing data. The primary approach to changing or removing data from a sequential-access file is to create a new one, from the old one, with the updated data. Because of the updating ease provided with random-access files, this chapter concentrates on creating, reading, and adding to sequential files.

Opening and Closing Files

Before you can create, write to, or read from a disk file, you must open the file. This is analogous to opening a filing cabinet before working with a file stored in the cabinet. Once you are done with a cabinet's file, you close the file drawer. You also must close a disk file when you finish with it.

When you open a disk file, you only have to inform C++ of the filename and what you want to do (write to, add to, or read from). C++ and your operating system work together to make sure the disk is ready and to create an entry in your file directory (if you are creating a file) for the filename. When you close a file, C++ writes any remaining data to the file, releases the file from the program, and updates the file directory to reflect the file's new size.

CAUTION: You must ensure that the FILES= statement in your CONFIG.SYS file is large enough to hold the maximum number of disk files you have open, with one left for your C++ program. If you are unsure how to do this, check your DOS reference manual or a beginner's book about DOS.

To open a file, you must call the `open()` function. To close a file, call the `close()` function. Here is the format of these two function calls:


```
file_ptr.open(file_name, access);
```

and

```
file_ptr.close();
```

`file_ptr` is a special type of pointer that only points to files, not data variables.

Your operating system handles the exact location of your data in the disk file. You don't want to worry about the exact track and sector number of your data on the disk. Therefore, you let `file_ptr` point to the data you are reading and writing. Your program only has to generically manage `file_ptr`, whereas C++ and your operating system take care of locating the actual physical data.

`file_name` is a string (or a character pointer that points to a string) containing a valid filename for your computer. `file_name` can contain a complete disk and directory pathname. You can specify the filename in uppercase or lowercase letters.

`access` must be one of the values from Table 30.1.

Table 30.1. Possible access modes.

Mode	*Description*
`app`	Open the file for appending (adding to it).
`ate`	Seek to end of file on opening it.
`in`	Open the file for reading.
`out`	Open the file for writing.
`binary`	Open the file in binary mode.
`trunc`	Discard contents if file exists
`nocreate`	If file doesn't exist, open fails.
`noreplace`	If file exists, open fails unless appending or seeking to end of file on opening.

The default access mode for file access is a *text mode.* A text file is an ASCII file, compatible with most other programming languages and applications. Text files do not always contain text, in the word-processing sense of the word. Any data you have to store can go in a text file. Programs that read ASCII files can read data you create as C++ text files. For a discussion of binary file access, see the box that follows.

Binary Modes

If you specify binary access, C++ creates or reads the file in a binary format. Binary data files are "squeezed"—they take less space than text files. The disadvantage of using binary files is that other programs cannot always read the data files. Only C++ programs written to access binary files can read and write to them. The advantage of binary files is that you save disk space because your data files are more compact. Other than the access mode in the `open()` function, you use no additional commands to access binary files with your C++ programs.

The binary format is a system-specific file format. In other words, not all computers can read a binary file created on another computer.

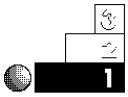
If you open a file for writing, C++ creates the file. If a file by that name already exists, C++ overwrites the old file with no warning. You must be careful when opening files so you do not overwrite existing data that you want to save.

If an error occurs during the opening of a file, C++ does not create a valid file pointer. Instead, C++ creates a file pointer equal to zero. For example, if you open a file for output, but use a disk name that is invalid, C++ cannot open the file and makes the file pointer equal to zero. Always check the file pointer when writing disk file programs to ensure the file opened properly.

TIP: Beginning programmers like to open all files at the beginning of their programs and close them at the end. This is not always the best method. Open files immediately before you access them and close them immediately when you are done with them. This habit protects the files because they are closed immediately after you are done with them. A closed file is more likely to be protected in the unfortunate (but possible) event of a power failure or computer breakdown.

This section contains much information on file-access theories. The following examples help illustrate these concepts.

Examples



1. Suppose you want to create a file for storing your house payment records for the previous year. Here are the first few lines in the program which creates a file called HOUSE.DAT on your disk:

```
#include <fstream.h>

main()
{
   ofstream file_ptr;   // Declares a file pointer for writing
   file_ptr.open("house.dat", ios::out);  // Creates the file
```

The remainder of the program writes data to the file. The program never has to refer to the filename again. The program uses the `file_ptr` variable to refer to the file. Examples in the next few sections illustrate how. There is nothing special about `file_ptr`, other than its name (although the name is meaningful in this case). You can name file pointer variables `XYZ` or `a908973` if you like, but these names would not be meaningful.

You must include the fstream.h header file because it contains the definition for the `ofstream` and `ifstream` declarations. You don't have to worry about the physical specifics. The `file_ptr` "points" to data in the file as you write it. Put the declarations in your programs where you declare other variables and arrays.

TIP: Because files are not part of your program, you might find it useful to declare file pointers globally. Unlike data in variables, there is rarely a reason to keep file pointers local.

Before finishing with the program, you should close the file. The following `close()` function closes the house file:

```
file_ptr.close();      // Close the house payment file.
```

2. If you want, you can put the complete pathname in the file's name. The following opens the household payment file in a subdirectory on the D: disk drive:

```
file_ptr.open("d:\mydata\house.dat", ios::out);
```

3. If you want, you can store a filename in a character array or point to it with a character pointer. Each of the following sections of code is equivalent:

```
char    fn[ ] = "house.dat";  // Filename in character array.
file_ptr.open(fn, ios::out);   // Creates the file.

char    *myfile = "house.dat";   // Filename pointed to.
file_ptr.open(myfile, ios::out);   // Creates the file.

// Let the user enter the filename.
cout << "What is the name of the household file? ";
gets(filename); // Filename must be an array or
                // character pointer.
file_ptr.open(filename, ios::out);   // Creates the file.
```

No matter how you specify the filename when opening the file, close the file with the file pointer. This `close()` function closes the open file, no matter which method you used to open the file:

```
file_ptr.close();   // Close the house payment file.
```

4. You should check the return value from `open()` to ensure the file opened properly. Here is code after `open()` that checks for an error:


```
#include <fstream.h>

main()
{
   ofstream file_ptr;   // Declares a file pointer.
```

```
file_ptr.open("house.dat", ios::out); // Creates the file.
if (!file_ptr)
{  cout << "Error opening file.\n"; }
else
{
   // Rest of output commands go here.
```

5. You can open and write to several files in the same program. Suppose you wanted to read data from a payroll file and create a backup payroll data file. You have to open the current payroll file using the `in` reading mode, and the backup file in the output `out` mode.

   For each open file in your program, you must declare a different file pointer. The file pointers used by your input and output statement determine on which file they operate. If you have to open many files, you can declare an array of file pointers.

   Here is a way you can open the two payroll files:

```
#include <fstream.h>

ifstream       file_in;       // Input file
ofstream       file_out;      // Output file

main()
{
   file_in.open("payroll.dat", ios::in);   // Existing file
   file_out.open("payroll.BAK", ios::out); // New file
```

   When you finish with these files, be sure to close them with these two `close()` function calls:

```
file_in.close();
file_out.close();
```

Writing to a File

Any input or output function that requires a device performs input and output with files. You have seen most of these already. The most common file I/O functions are

```
get() and put()
gets() and puts()
```

You also can use `file_ptr` as you do with `cout` or `cin`.

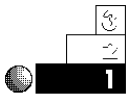
The following function call reads three integers from a file pointed to by `file_ptr`:

```
file_ptr >> num1 >> num2 >> num3;    // Reads three variables.
```

There is always more than one way to write data to a disk file. Most the time, more than one function will work. For example, if you write many names to a file, both `puts()` and `file_ptr <<` work. You also can write the names using `put()`. You should use whichever function you are most comfortable with. If you want a newline character (`\n`) at the end of each line in your file, the `file_ptr <<` and `puts()` are probably easier than `put()`, but all three will do the job.

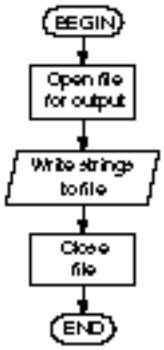
TIP: Each line in a file is called a *record*. By putting a newline character at the end of file records, you make the input of those records easier.

Examples



1. The following program creates a file called NAMES.DAT. The program writes five names to a disk file using `file_ptr <<`.

```
// Filename: C3OWR1.CPP
// Writes five names to a disk file.
#include <fstream.h>

ofstream fp;
```




```
void main()
{
   fp.open("NAMES.DAT", ios::out);  // Creates a new file.

   fp << "Michael Langston\n";
   fp << "Sally Redding\n";
   fp << "Jane Kirk\n";
   fp << "Stacy Wikert\n";
   fp << "Joe Hiquet\n";
   fp.close();   // Release the file.
   return;
}
```

To keep this first example simple, error checking was not done on the `open()` function. The next few examples check for the error.

NAMES.TXT is a text data file. If you want, you can read this file into your word processor (use your word processor's command for reading ASCII files) or use the MS-DOS TYPE command (or your operating system's equivalent command) to display this file on-screen. If you were to display NAMES.TXT, you would see:

```
Michael Langston
Sally Redding
Jane Kirk
Stacy Wikert
Joe Hiquet
```

2. The following file writes the numbers from 1 to 100 to a file called NUMS.1.

```
// Filename: C30WR2.CPP
// Writes 1 to 100 to a disk file.

#include <fstream.h>

ofstream       fp;

void main()
```

```
{
   int ctr;

   fp.open("NUMS.1", ios::out);   // Creates a new file.
   if (!fp)
     {  cout << "Error opening file.\n"; }
   else
     {
        for (ctr = 1; ctr < 101; ctr++)
           {  fp << ctr << " "; }
     }
   fp.close();
   return;
}
```

The numbers are not written one per line, but with a space between each of them. The format of the `file_ptr <<` determines the format of the output data. When writing data to disk files, keep in mind that you have to read the data later. You have to use "mirror-image" input functions to read data you output to files.

Writing to a Printer

Functions such as `open()` and others were not designed to write only to files. They were designed to write to any device, including files, the screen, and the printer. If you must write data to a printer, you can treat the printer as if it were a file. The following program opens a file pointer using the MS-DOS name for a printer located at LPT1 (the MS-DOS name for the first parallel printer port):

```
// Filename: C30PRNT.CPP
// Prints to the printer device

#include <fstream.h>

ofstream prnt;   // Points to the printer.

void main()
```

```
{
   prnt.open("LPT1", ios::out);
   prnt << "Printer line 1\n";     // 1st line printed.
   prnt << "Printer line 2\n";     // 2nd line printed.
   prnt << "Printer line 3\n";     // 3rd line printed.
   prnt.close();
return;
}
```

Make sure your printer is on and has paper before you run this program. When you run the program, you see this printed on the printer:

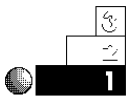
```
Printer line 1
Printer line 2
Printer line 3
```

Adding to a File

You can easily add data to an existing file or create new files, by opening the file in append access mode. Data files on the disk are rarely static; they grow almost daily due to (hopefully!) increased business. Being able to add to data already on the disk is very useful, indeed.

Files you open for append access (using `ios::app`) do not have to exist. If the file exists, C++ appends data to the end of the file when you write the data. If the file does not exist, C++ creates the file (as is done when you open a file for write access).

Example



The following program adds three more names to the NAMES.DAT file created in an earlier example.

```
// Filename: C30AP1.CPP
// Adds three names to a disk file.

#include <fstream.h>
```

```
ofstream    fp;

void main()
{
   fp.open("NAMES.DAT", ios::app);   // Adds to file.
   fp << "Johnny Smith\n";
   fp << "Laura Hull\n";
   fp << "Mark Brown\n";
   fp.close();                  // Release the file.
   return;
}
```

Here is what the file now looks like:

```
Michael Langston
Sally Redding
Jane Kirk
Stacy Wikert
Joe Hiquet
Johnny Smith
Laura Hull
Mark Brown
```

NOTE: If the file does not exist, C++ creates it and stores the three names to the file.

Basically, you only have to change the `open()` function's access mode to turn a file-creation program into a file-appending program.

Reading from a File

Files must exist prior to opening them for read access.

Once the data is in a file, you must be able to read that data. You must open the file in a read access mode. There are several ways to read data. You can read character data one character at a time or one string at a time. The choice depends on the format of the data.

Files you open for read access (using `ios::in`) must exist already, or C++ gives you an error. You cannot read a file that does not exist. `open()` returns zero if the file does not exist when you open it for read access.

Another event happens when reading files. Eventually, you read all the data. Subsequent reading produces errors because there is no more data to read. C++ provides a solution to the end-of-file occurrence. If you attempt to read from a file that you have completely read the data from, C++ returns the value of zero. To find the end-of-file condition, be sure to check for zero when reading information from files.

Examples

1. This program asks the user for a filename and prints the contents of the file to the screen. If the file does not exist, the program displays an error message.

```
// Filename: C30RE1.CPP
// Reads and displays a file.

#include <fstream.h>
#include <stdlib.h>

ifstream fp;

void main()
{
   char filename[12]; // Holds user's filename.
   char in_char;      // Input character.

   cout << "What is the name of the file you want to see? ";
   cin >> filename;
   fp.open(filename, ios::in);
   if (!fp)
   {
     cout << "\n\n*** That file does not exist ***\n";
     exit(0);    // Exit program.
   }
   while (fp.get(in_char))
     {  cout << in_char;  }
   fp.close();
   return;
}
```

Here is the resulting output when the NAMES.DAT file is requested:

```
What is the name of the file you want to see? NAMES.DAT
Michael Langston
Sally Redding
Jane Kirk
Stacy Wikert
Joe Hiquet
Johnny Smith
Laura Hull
Mark Brown
```

Because newline characters are in the file at the end of each name, the names appear on-screen, one per line. If you attempt to read a file that does not exist, the program displays the following message:

```
*** That file does not exist ***
```

2. This program reads one file and copies it to another. You might want to use such a program to back up important data in case the original file is damaged.

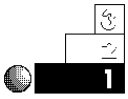
   The program must open two files, the first for reading, and the second for writing. The file pointer determines which of the two files is being accessed.

```
// Filename: C3ORE2.CPP
// Makes a copy of a file.

#include <fstream.h>
#include <stdlib.h>

ifstream in_fp;
ofstream out_fp;

void main()
{
   char in_filename[12];   // Holds original filename.
   char out_filename[12];  // Holds backup filename.
   char in_char;           // Input character.
```

```
   cout << "What is the name of the file you want to back up?
";
   cin >> in_filename;
   cout << "What is the name of the file ";
   cout << "you want to copy " << in_filename << " to? ";
   cin >> out_filename;
   in_fp.open(in_filename, ios::in);
   if (!in_fp)
   {
     cout << "\n\n*** " << in_filename << " does not exist
***\n";
     exit(0);   // Exit program
   }
   out_fp.open(out_filename, ios::out);
   if (!out_fp)
   {
      cout << "\n\n*** Error opening " << in_filename << "
***\n";
      exit(0);    // Exit program
   }
   cout << "\nCopying...\n";   // Waiting message.
   while (in_fp.get(in_char))
     { out_fp.put(in_char); }
   cout << "\nThe file is copied.\n";
   in_fp.close();
   out_fp.close();
   return;
}
```

Review Questions

Answers to the review questions are in Appendix B.

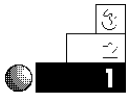


1. What are the three ways to access sequential files?
2. What advantage do disk files have over holding data in memory?
3. How do sequential files differ from random-access files?

4. What happens if you open a file for read access and the file does not exist?
5. What happens if you open a file for write access and the file already exists?
6. What happens if you open a file for append access and the file does not exist?

7. How does C++ inform you that you have reached the end-of-file condition?

Review Exercises



1. Write a program that creates a file containing the following data:

   Your name
   Your address
   Your phone number
   Your age

2. Write a second program that reads and prints the data file you created in Exercise 1.
3. Write a program that takes your data created in Exercise 1 and writes it to the screen one word per line.

4. Write a program for PCs that backs up two important files: the AUTOEXEC.BAT and CONFIG.SYS. Call the backup files AUTOEXEC.SAV and CONFIG.SAV.

5. Write a program that reads a file and creates a new file with the same data, except reverse the case on the second file. Everywhere uppercase letters appear in the first file, write lowercase letters to the new file, and everywhere lowercase letters appear in the first file, write uppercase letters to the new file.

Summary

You can now perform one of the most important requirements of data processing: writing and reading to and from disk files. Before this chapter, you could only store data in variables. The short life of variables (they only last as long as your program is running) made long-term storage of data impossible. You can now save large amounts of data in disk files to process later.

Reading and writing sequential files involves learning more concepts than actual commands or functions. The `open()` and `close()` functions are the most important functions you learned in this chapter. You are now familiar with most of the I/O functions needed to retrieve data to and from disk files.

The next chapter concludes the discussion of disk files in this book. You will learn how to create and use random-access files. By programming with random file access, you can read selected data from a file, as well as change data without having to rewrite the entire file.

Random-Access Files

This chapter introduces the concept of random file access. Random file access enables you to read or write any data in your disk file without having to read or write every piece of data before it. You can quickly search for, add, retrieve, change, and delete information in a random-access file. Although you need a few new functions to access files randomly, you find that the extra effort pays off in flexibility, power, and speed of disk access.

This chapter introduces

- Random-access files
- File records
- The `seekg()` function
- Special-purpose file I/O functions

With C++'s sequential and random-access files, you can do everything you would ever want to do with disk data.

Random File Records

Random files exemplify the power of data processing with C++. Sequential file processing is slow unless you read the entire file into arrays and process them in memory. As explained in Chapter 30, however, you have much more disk space than RAM, and most disk files do not even fit in your RAM at one time. Therefore, you need a way to quickly read individual pieces of data from a file in any order and process them one at a time.

A record to a file is like a structure to variables.

Generally, you read and write file *records*. A record to a file is analogous to a C++ structure. A record is a collection of one or more data values (called *fields*) you read and write to disk. Generally, you store data in structures and write the structures to disk where they are called records. When you read a record from disk, you generally read that record into a structure variable and process it with your program.

Unlike most programming languages, not all disk data for C++ programs has to be stored in record format. Typically, you write a stream of characters to a disk file and access that data either sequentially or randomly by reading it into variables and structures.

The process of randomly accessing data in a file is simple. Think about the data files of a large credit card organization. When you make a purchase, the store calls the credit card company to receive authorization. Millions of names are in the credit card company's files. There is no quick way the credit card company could read every record sequentially from the disk that comes before yours. Sequential files do not lend themselves to quick access. It is not feasible, in many situations, to look up individual records in a data file with sequential access.

The credit card companies must use a random file access so their computers can go directly to your record, just as you go directly to a song on a compact disk or record album. The functions you use are different from the sequential functions, but the power that results from learning the added functions is worth the effort.

You do not have to rewrite an entire file to change random-access file data.

When your program reads and writes files randomly, it treats the file like a big array. With arrays, you know you can add, print, or remove values in any order. You do not have to start at the first

array element, sequentially looking at the next one, until you get the element you need. You can view your random-access file in the same way, accessing the data in any order.

Most random file records are *fixed-length* records. Each record (usually a row in the file) takes the same amount of disk space. Most of the sequential files you read and wrote in the previous chapters were variable-length records. When you are reading or writing sequentially, there is no need for fixed-length records because you input each value one character, word, string, or number at a time, and look for the data you want. With fixed-length records, your computer can better calculate where on the disk the desired record is located.

Although you waste some disk space with fixed-length records (because of the spaces that pad some of the fields), the advantages of random file access compensate for the "wasted" disk space (when the data do not actually fill the structure size).

TIP: With random-access files, you can read or write records in any order. Therefore, even if you want to perform sequential reading or writing of the file, you can use random-access processing and "randomly" read or write the file in sequential record number order.

Opening Random-Access Files

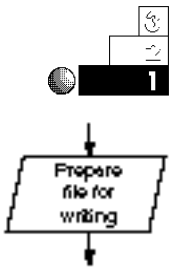
Just as with sequential files, you must open random-access files before reading or writing to them. You can use any of the read access modes mentioned in Chapter 30 (such as `ios::in`) only to read a file randomly. However, to modify data in a file, you must open the file in one of the update modes, repeated for you in Table 31.1.

Table 31.1. Random-access update modes.

Mode	*Description*
`app`	Open the file for appending (adding to it)
`ate`	Seek to end of file on opening it
`in`	Open file for reading
`out`	Open file for writing
`binary`	Open file in binary mode
`trunc`	Discard contents if file exists
`nocreate`	If file doesn't exist, open fails
`noreplace`	If file exists, open fails unless appending or seeking to end of file on opening

There is really no difference between sequential files and random files in C++. The difference between the files is not physical, but lies in the method you use to access them and update them.

Examples



1. Suppose you want to write a program to create a file of your friends' names. The following `open()` function call suffices, assuming `fp` is declared as a file pointer:

```
fp.open("NAMES.DAT", ios::out);
if (!fp)
  { cout << "\n*** Cannot open file ***\n"; }
```

No update `open()` access mode is needed if you are only creating the file. However, what if you wanted to create the file, write names to it, and give the user a chance to change any of the names before closing the file? You then have to open the file like this:

```
fp.open("NAMES.DAT", ios::in | ios::out);
if (!fp)
   cout << "\n*** Cannot open file ***\n";
```

This code enables you to create the file, then change data you wrote to the file.

2. As with sequential files, the only difference between using a binary `open()` access mode and a text mode is that the file you create is more compact and saves disk space. You cannot, however, read that file from other programs as an ASCII text file. The previous `open()` function can be rewritten to create and allow updating of a binary file. All other file-related commands and functions work for binary files just as they do for text files.

```
fp.open("NAMES.DAT", ios::in | ios::out | ios::binary);
if (!fp)
   cout << "\n*** Cannot open file ***\n";
```

The `seekg()` Function

C++ provides a function that enables you to read to a specific point in a random-access data file. This is the `seekg()` function. The format of `seekg()` is

```
file_ptr.seekg(long_num, origin);
```

You can read forwards or backwards from any point in the file with `seekg()`.

`file_ptr` is the pointer to the file that you want to access, initialized with an `open()` statement. `long_num` is the number of bytes in the file you want to skip. C++ does not read this many bytes, but literally skips the data by the number of bytes specified in `long_num`. Skipping the bytes on the disk is much faster than reading them. If `long_num` is negative, C++ skips backwards in the file (this allows for rereading of data several times). Because data files can be large, you must declare `long_num` as a long integer to hold a large amount of bytes.

`origin` is a value that tells C++ where to begin the skipping of bytes specified by `long_num`. `origin` can be any of the three values shown in Table 31.2.

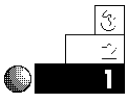
Table 31.2. Possible **origin** values.

Description	origin	*Equivalent*
Beginning of file	SEEK_SET	ios::beg
Current file position	SEEK_CUR	ios::cur
End of file	SEEK_END	ios::end

The origins SEEK_SET, SEEK_CUR, and SEEK_END are defined in stdio.h. The equivalents `ios::beg`, `ios::cur`, and `ios::end` are defined in fstream.h.

NOTE: Actually, the file pointer plays a much more important role than simply "pointing to the file" on the disk. The file pointer continually points to the exact location of the *next byte to read or write*. In other words, as you read data from either a sequential or random-access file, the file pointer increments with each byte read. By using `seekg()`, you can move the file pointer forward or backward in the file.

Examples



1. No matter how far into a file you have read, the following `seekg()` function positions the file pointer back to the beginning of a file:

   ```
   fp.seekg(0L, SEEK_SET); // Position file pointer at beginning.
   ```

   The constant `0L` passes a long integer `0` to the `seekg()` function. Without the `L`, C++ passes a regular integer and this does not match the prototype for `seekg()` that is located in fstream.h. Chapter 4, "Variables and Literals," explained the use of data type suffixes on numeric constants, but the suffixes have not been used until now.

   This `seekg()` function literally reads "move the file pointer 0 bytes from the beginning of the file."

2. The following example reads a file named MYFILE.TXT twice, once to send the file to the screen and once to send the file to the printer. Three file pointers are used, one for each device (the file, the screen, and the printer).

```
// Filename: C31TWIC.CPP
// Writes a file to the printer, rereads it,
// and sends it to the screen.

#include <fstream.h>
#include <stdlib.h>
#include <stdio.h>

ifstream in_file;   // Input file pointer.
ofstream scrn;      // Screen pointer.
ofstream prnt;      // Printer pointer.

void main()
{
   char in_char;

   in_file.open("MYFILE.TXT", ios::in);
   if (!in_file)
   {
      cout << "\n*** Error opening MYFILE.TXT ***\n";
      exit(0);
   }
   scrn.open("CON", ios::out);    // Open screen device.
   while (in_file.get(in_char))
   { scrn << in_char; } // Output characters to the screen.
   scrn.close();  // Close screen because it is no
                  // longer needed.
   in_file.seekg(0L, SEEK_SET);   // Reposition file pointer.
   prnt.open("LPT1", ios::out);   // Open printer device.
   while (in_file.get(in_char))
     { prnt << in_char; }   // Output characters to the
                            // printer.
   prnt.close();          // Always close all open files.
   in_file.close();
   return;
}
```

You also can close then reopen a file to position the file pointer at the beginning, but using `seekg()` is a more efficient method.

Of course, you could have used regular I/O functions to write to the screen, rather than having to open the screen as a separate device.

3. The following `seekg()` function positions the file pointer at the 30th byte in the file. (The next byte read is the 31st byte.)

```
file_ptr.seekg(30L, SEEK_SET);   // Position file pointer
                                 // at the 30th byte.
```

This `seekg()` function literally reads "move the file pointer 30 bytes from the beginning of the file."

If you write structures to a file, you can quickly seek any structure in the file using the `sizeof()` function. Suppose you want the 123rd occurrence of the structure tagged with `inventory`. You would search using the following `seekg()` function:

```
file_ptr.seekg((123L * sizeof(struct inventory)), SEEK_SET);
```

4. The following program writes the letters of the alphabet to a file called ALPH.TXT. The `seekg()` function is then used to read and display the ninth and 17th letters (*I* and *Q*).

```
// Filename: C31ALPH.CPP
// Stores the alphabet in a file, then reads
// two letters from it.

#include <fstream.h>
#include <stdlib.h>
#include <stdio.h>

fstream fp;

void main()
{
   char ch;     // Holds A through Z.
```

```
    // Open in update mode so you can read file after writing to it.
    fp.open("alph.txt", ios::in | ios::out);
    if (!fp)
    {
      cout << "\n*** Error opening file ***\n";
      exit(0);
    }
    for (ch = 'A'; ch <= 'Z'; ch++)
    { fp << ch;  }  // Write letters.
    fp.seekg(8L, ios::beg);   // Skip eight letters, point to I.
    fp >> ch;
    cout << "The first character is " << ch << "\n";
    fp.seekg(16L, ios::beg);   // Skip 16 letters, point to Q.
    fp >> ch;
    cout << "The second character is " << ch << "\n";
    fp.close();
    return;
}
```

5. To point to the end of a data file, you can use the `seekg()` function to position the file pointer at the last byte. Subsequent `seekg()`s should then use a negative `long_num` value to skip backwards in the file. The following `seekg()` function makes the file pointer point to the end of the file:

```
file_ptr.seekg(0L, SEEK_END); // Position file
                              // pointer at the end.
```

   This `seekg()` function literally reads "move the file pointer 0 bytes from the end of the file." The file pointer now points to the end-of-file marker, but you can `seekg()` backwards to find other data in the file.

6. The following program reads the ALPH.TXT file (created in Exercise 4) backwards, printing each character as it skips back in the file.

```
// Filename: C31BACK.CPP
// Reads and prints a file backwards.
```

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

ifstream fp;

void main()
{
   int ctr;   // Steps through the 26 letters in the file.
   char in_char;

   fp.open("ALPH.TXT", ios::in);
   if (!fp)
   {
     cout << "\n*** Error opening file ***\n";
     exit(0);
   }
   fp.seekg(-1L, SEEK_END);   // Point to last byte in
                              // the file.
   for (ctr = 0; ctr < 26; ctr++)
   {
      fp >> in_char;
      fp.seekg(-2L, SEEK_CUR);
      cout << in_char;
   }
   fp.close();
   return;
}
```

This program also uses the `SEEK_CUR` `origin` value. The last `seekg()` in the program seeks two bytes backwards from the *current position*, not the beginning or end as the previous examples have. The `for` loop towards the end of the program performs a "skip-two-bytes-back, read-one-byte-forward" method to skip through the file backwards.

7. The following program performs the same actions as Example 4 (C31ALPH.CPP), with one addition. When the letters *I* and *Q* are found, the letter *x* is written over the *I* and *Q*. The `seekg()` must be used to back up one byte in the file to overwrite the letter just read.

```
// Filename: C31CHANG.CPP
// Stores the alphabet in a file, reads two letters from it,
// and changes those letters to xs.

#include <fstream.h>
#include <stdlib.h>
#include <stdio.h>

fstream fp;

void main()
{
   char ch;   // Holds A through Z.

// Open in update mode so you can read file after writing to it.
   fp.open("alph.txt", ios::in | ios::out);
   if (!fp)
   {
     cout << "\n*** Error opening file ***\n";
     exit(0);
   }
   for (ch = 'A'; ch <= 'Z'; ch++)
      { fp << ch; }   // Write letters
   fp.seekg(8L, SEEK_SET);  // Skip eight letters, point to I.
   fp >> ch;
   // Change the Q to an x.
   fp.seekg(-1L, SEEK_CUR);
   fp << 'x';
   cout << "The first character is " << ch << "\n";
   fp.seekg(16L, SEEK_SET);   // Skip 16 letters, point to Q.
   fp >> ch;
   cout << "The second character is " << ch << "\n";
   // Change the Q to an x.
   fp.seekg(-1L, SEEK_CUR);
   fp << 'x';
   fp.close();
   return;
}
```

The file named ALPH.TXT now looks like this:

```
ABCDEFGHxJKLMNOPxRSTUVWXYZ
```

This program forms the basis of a more complete data file management program. After you master the `seekg()` functions and become more familiar with disk data files, you will begin to write programs that store more advanced data structures and access them.

The mailing list application in Appendix F is a good example of what you can do with random file access. The user is given a chance to change names and addresses already in the file. The program, using random access, seeks for and changes selected data without rewriting the entire disk file.

Other Helpful I/O Functions

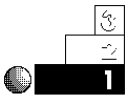
There are several more disk I/O functions available that you might find useful. They are mentioned here for completeness. As you perform more powerful disk I/O, you might find a use for many of these functions. Each of these functions is prototyped in the fstream.h header file.

- `read(array, count)`: Reads the data specified by `count` into the array or pointer specified by `array`. `read()` is called a *buffered I/O* function. `read()` enables you to read much data with a single function call.
- `write(array, count)`: Writes `count array` bytes to the specified file. `write()` is a buffered I/O function. `write()` enables you to write much data in a single function call.
- `remove(filename)`: Erases the file named by `filename`. `remove()` returns a `0` if the file was erased successfully and `-1` if an error occurred.

Many of these (and other built-in I/O functions that you learn in your C++ programming career) are helpful functions that you could duplicate using what you already know.

The buffered I/O file functions enable you to read and write entire arrays (including arrays of structures) to the disk in a single function call.

Examples



1. The following program requests a filename from the user and erases the file from the disk using the `remove()` function.

```
// Filename: C31ERAS.CPP
// Erases the file specified by the user.

#include <stdio.h>
#include <iostream.h>

void main()
{
   char filename[12];

   cout << "What is the filename you want me to erase? ";
   cin >> filename;
   if (remove(filename) == -1)
   {  cout << "\n*** I could not remove the file ***\n"; }
   else
   {  cout << "\nThe file " << filename << " is now removed\n";}
   return;
}
```

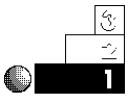
2. The following function is part of a larger program that receives inventory data, in an array of structures, from the user. This function is passed the array name and the number of elements (structure variables) in the array. The `write()` function then writes the complete array of structures to the disk file pointed to by `fp`.

```
void write_str(inventory items[ ], int inv_cnt)
{
   fp.write(items, inv_cnt * sizeof(inventory);
   return;
}
```

If the inventory array had 1,000 elements, this one-line function would still write the entire array to the disk file. You could use the `read()` function to read the entire array of structures from the disk in a single function call.

Review Questions

The answers to the review questions are in Appendix B.

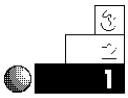


1. What is the difference between records and structures?
2. True or false: You have to create a random-access file before reading from it randomly.
3. What happens to the file pointer as you read from a file?

4. What are the two buffered file I/O functions?
5. What is wrong with this program?

```
#include <fstream.h>
ifstream  fp;
void main()
{
   char in_char;
   fp.open(ios::in | ios::binary);
   if (fp.get(in_char))
   {  cout << in_char;  }   // Write to the screen
   fp.close();
   return;
}
```

Review Exercises



1. Write a program that asks the user for a list of five names, then writes the names to a file. Rewind the file and display its contents on-screen using the `seekg()` and `get()` functions.

2. Rewrite the program in Exercise 1 so it displays every other character in the file of names.

3. Write a program that reads characters from a file. If the input character is a lowercase letter, change it to uppercase. If the input character is an uppercase letter, change it to lowercase. Do not change other characters in the file.

4. Write a program that displays the number of nonalphabetic characters in a file.

5. Write a grade-keeping program for a teacher. Allow the teacher to enter up to 10 students' grades. Each student has three grades for the semester. Store the students' names and their three grades in an array of structures and store the data on the disk. Make the program menu-driven. Include options of adding more students, viewing the file's data, or printing the grades to the printer with a calculated class average.

Summary

C++ supports random-access files with several functions. These functions include error checking, file pointer positioning, and the opening and closing of files. You now have the tools you need to save your C++ program data to disk for storage and retrieval.

The mailing-list application in Appendix F offers a complete example of random-access file manipulation. The program enables the user to enter names and addresses, store them to disk, edit them, change them, and print them from the disk file. The mailing-list program combines almost every topic from this book into a complete application that "puts it all together."

Introduction to Object-Oriented Programming

The most widely used object-oriented programming language today is C++. C++ provides *classes*—which are its objects. Classes really distinguish C++ from C. In fact, before the name C++ was coined, the C++ language was called "C with classes."

This chapter attempts to expose you to the world of object-oriented programming, often called *OOP*. You will probably not become a master of OOP in these few short pages, however, you are ready to begin expanding your C++ knowledge.

This chapter introduces the following concepts:

- C++ classes
- Member functions
- Constructors
- Destructors

This chapter concludes your introduction to the C++ language. After mastering the techniques taught in this book, you will be ready to modify the mailing list program in Appendix F to suit your own needs.

What Is a Class?

A *class* is a user-defined data type that resembles a structure. A class can have data members, but unlike the structures you have seen thus far, classes can also have *member functions*. The data members can be of any type, whether defined by the language or by you. The member functions can manipulate the data, create and destroy class variables, and even redefine C++'s operators to act on the class objects.

Classes have several types of members, but they all fall into two categories: data members and member functions.

Data Members

Data members can be of any type. Here is a simple class:

```
// A sphere class.
class Sphere
{
public:
   float r;          // Radius of sphere
   float x, y, z;    // Coordinates of sphere
};
```

Notice how this class resembles structures you have already seen, with the exception of the `public` keyword. The `Sphere` class has four data members: `r`, `x`, `y`, and `z`. In this case, the `public` keyword plays an important role; it identifies the class `Sphere` as a structure. As a matter of fact, in C++, a *public class* is physically identical to a structure. For now, ignore the `public` keyword; it is explained later in this chapter.

Member Functions

A class can also have *member functions* (members of a class that manipulate data members). This is one of the primary features that distinguishes a class from a structure. Here is the `Sphere` class again, with member functions added:

```
#include  <math.h>

const float PI = 3.14159;
// A sphere class.
class Sphere
{
public:
   float r;        // Radius of sphere
   float x, y, z;  // Coordinates of sphere
   Sphere(float xcoord, float ycoord, float zcoord, float radius)
     { x = xcoord; y = ycoord; z = zcoord; r = radius; }
   ~Sphere() { }
   float volume()
   {
      return (r * r * r * 4 * PI / 3);
   }
   float surface_area()
   {
     return (r * r * 4 * PI);
   }
};
```

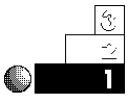
This `Sphere` class has four member functions: `Sphere()`, `~Sphere()`, `volume()`, and `surface_area()`. The class is losing its similarity to a structure. These member functions are very short. (The one with the strange name of `~Sphere()` has no code in it.) If the codes of the member functions were much longer, only the prototypes would appear in the class, and the code for the member functions would follow later in the program.

C++ programmers call class data *objects* because classes do more than simply hold data. Classes act on data; in effect, a class is an object that manipulates itself. All the data you have seen so far in this book is *passive* data (data that has been manipulated by code in the program). Classes' member functions actually manipulate class data.

Constructors create and initialize class data.

In this example, the class member `Sphere()` is a special function. It is a *constructor* function, and its name must always be the same as its class. Its primary use is declaring a new instance of the class.

Examples



1. The following program uses the `Sphere()` class to initialize a class variable (called a class *instance*) and print it.

```
// Filename: C32CON.CPP
// Demonstrates use of a class constructor function.

#include  <iostream.h>
const float PI = 3.14159;  // Approximate value of pi.

// A sphere class.
class Sphere
{
public:
   float r;          // Radius of sphere
   float x, y, z;  // Coordinates of sphere
   Sphere(float xcoord, float ycoord,
          float zcoord, float radius)
       { x = xcoord; y = ycoord; z = zcoord; r = radius; }
   ~Sphere() { }
   float volume()
   {
     return (r * r * r * 4 * PI / 3);
   }
   float surface_area()
   {
      return (r * r * 4 * PI);
   }
};

void main()
{
  Sphere s(1.0, 2.0, 3.0, 4.0);

   cout << "X = " << s.x << ", Y = " << s.y
        << ", Z = " << s.z << ", R = " << s.r << "\n";
   return;
}
```

Note: In OOP, the `main()` function (and all it calls) becomes smaller because member functions contain the code that manipulates all class data.

Indeed, this program looks different from those you have seen so far. This example is your first true exposure to OOP programming. Here is the output of this program:

```
X = 1, Y = 2, Z = 3, R = 4
```

This program illustrates the `Sphere()` constructor function. The constructor function is the only member function called by the program. Notice the `~Sphere()` member function constructed `s`, and initialized its data members as well.

Destructors erase class data.

The other special function is the *destructor* function, `~Sphere()`. Notice that it also has the same name as the class, but with a *tilde* (~) as a prefix. The destructor function never takes arguments, and never returns values. Also notice that this destructor doesn't do anything. Most destructors do very little. If a destructor has no real purpose, you do not have to specify it. When the class variable goes out of scope, the memory allocated for that class variable is returned to the system (in other words, an automatic destruction occurs). Programmers use destructor functions to free memory occupied by class data in advanced C++ applications.

Similarly, if a constructor doesn't serve any specific function, you aren't required to declare one. C++ allocates memory for a class variable when you define the class variable, just as it does for all other variables. As you learn more about C++ programming, especially when you begin using the advanced concept of *dynamic memory allocation*, constructors and destructors become more useful.

2. To illustrate that the `~Sphere()` destructor does get called (it just doesn't do anything), you can put a `cout` statement in the constructor as seen in the next program:

```
// Filename: C32DES.CPP
// Demonstrates use of a class destructor function.
```

```
#include  <iostream.h>
#include  <math.h>
const float PI = 3.14159;   // Approximate value of pi.

// A sphere class
class Sphere
{
public:
   float  r;        // Radius of sphere
   float  x, y, z;  // Coordinates of sphere
   Sphere(float xcoord, float ycoord,
          float zcoord, float radius)
   { x = xcoord; y = ycoord; z = zcoord; r = radius; }
   ~Sphere()
   {
    cout << "Sphere (" << x << ", " << y
         << ", " << z << ", " << r << ") destroyed\n";
   }
   float volume()
   {
      return (r * r * r * 4 * PI / 3);
   }
   float surface_area()
   {
     return (r * r * 4 * PI);
   }
};

void main(void)
{
   Sphere s(1.0, 2.0, 3.0, 4.0);
   // Construct a class instance.
   cout << "X = " << s.x << ", Y = "
        << s.y << ", Z = " << s.z << ", R = " << s.r << "\n";
   return;
}
```

Here is the output of this program:

```
X = 1, Y = 2, Z = 3, R = 4
Sphere (1, 2, 3, 4) destroyed
```

Notice that `main()` did not explicitly call the destructor function, but `~Sphere()` was called automatically when the class instance went out of scope.

3. The other member functions have been waiting to be used. The following program uses the `volume()` and `surface_area()` functions:

```
// Filename: C32MEM.CPP
// Demonstrates use of class member functions.

#include  <iostream.h>
#include  <math.h>
const float PI = 3.14159;  // Approximate value of pi.

// A sphere class.
class Sphere
{
public:
   float r;        // Radius of sphere
   float x, y, z;  // Coordinates of sphere
   Sphere(float xcoord, float ycoord,
          float zcoord, float radius)
   { x = xcoord; y = ycoord; z = zcoord; r = radius; }
   ~Sphere()
   {
      cout << "Sphere (" << x << ", " << y
           << ", " << z << ", " << r << ") destroyed\n";
   }
   float volume()
   {
       return (r * r * r * 4 * PI / 3);
   }
   float surface_area()
   {
     return (r * r * 4 * PI);
   }
};  // End of class.

void main()
{
   Sphere s(1.0, 2.0, 3.0, 4.0);
   cout << "X = " << s.x << ", Y = " << s.y
        << ", Z = " << s.z << ", R = " << s.r << "\n";
```

```
    cout << "The volume is " << s.volume() << "\n";
    cout << "The surface area is "
         << s.surface_area() << "\n";
}
```

The `volume()` and `surface_area()` functions could have been made *in-line*. This means that the compiler embeds the functions in the code, rather than calling them as functions. In C32MEM.CPP, there is essentially a separate function that is called using the data in `Sphere()`. By making it in-line, `Sphere()` essentially becomes a macro and is expanded in the code.

4. In the following program, `volume()` has been changed to an in-line function, creating a more efficient program:

```
// Filename: C32MEM1.CPP
// Demonstrates use of in-line class member functions.

#include  <iostream.h>
#include  <math.h>
const float PI = 3.14159;  // Approximate value of pi.

// A sphere class.
class Sphere
{
public:
   float r;        // Radius of sphere
   float x, y, z;  // Coordinates of sphere
   Sphere(float xcoord, float ycoord, float zcoord, float radius)
   { x = xcoord; y = ycoord; z = zcoord; r = radius; }
   ~Sphere()
   {
     cout << "Sphere (" << x << ", " << y
          << ", " << z << ", " << r << ") destroyed\n";
   }
   inline float volume()
   {
     return (r * r * r * 4 * PI / 3);
   }
   float surface_area()
   {
      return (r * r * 4 * PI);
```

```
   }
};

void main()
{
   Sphere s(1.0, 2.0, 3.0, 4.0);
   cout << "X = " << s.x << ", Y = " << s.y
        << ", Z = " << s.z << ", R = " << s.r << "\n";
   cout << "The volume is " << s.volume() << "\n";
   cout << "The surface area is " << s.surface_area() << "\n";
}
```

The inline functions expand to look like this to the compiler:

```
// C32MEM1A.CPP
// Demonstrates use of in-line class member functions.

#include  <iostream.h>
#include  <math.h>
const float PI = 3.14159;  // Approximate value of pi.

// A sphere class
class Sphere
{
public:
   float r;        // Radius of sphere
   float x, y, z;  // Coordinates of sphere
   Sphere(float xcoord, float ycoord, float zcoord, float radius)
   { x = xcoord; y = ycoord; z = zcoord; r = radius; }
   ~Sphere()
   {
      cout << "Sphere (" << x << ", " << y
           << ", " << z << ", " << r << ") destroyed\n";
   }
   inline float volume()
   {
      return (r * r * r * 4 * PI / 3);
   }
   float surface_area()
   {
     return (r * r * 4 * PI);
   }
};
```

```
void main()
{
   Sphere s(1.0, 2.0, 3.0, 4.0);
   cout << "X = " << s.x << ", Y = " << s.y
        << ", Z = " << s.z << ", R = " << s.r << "\n";
   cout << "The volume is " << (s.r * s.r * s.r * 4 * PI / 3)
        << "\n";
   cout << "The surface area is " << s.surface_area() << "\n";
}
```

The advantage of using in-line functions is that they execute faster—there's no function-call overhead involved because no function is actually called. The disadvantage is that if your functions are used frequently, your programs become larger and larger as functions are expanded.

Default Member Arguments

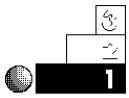
You can also give member functions arguments by default. Assume by default that the *y* coordinate of a sphere will be 2.0, the *z* coordinate will be 2.5, and the radius will be 1.0. Rewriting the previous example's constructor function to do this results in this code:

```
Sphere(float xcoord, float ycoord = 2.0, float zcoord = 2.5,
       float radius = 1.0)
  { x = xcoord; y = ycoord; z = zcoord; r = radius; }
```

You can create a sphere with the following instructions:

```
Sphere s(1.0);                  // Use all default

Sphere t(1.0, 1.1);             // Override y coord

Sphere u(1.0, 1.1, 1.2);        // Override y and z

Sphere v(1.0, 1.1, 1.2, 1.3);   // Override all defaults
```

Examples



1. Default arguments are used in the following code.

```
// Filename: C32DEF.CPP
// Demonstrates use of default arguments in
// class member functions.

#include  <iostream.h>
#include  <math.h>
const float PI = 3.14159;  // Approximate value of pi.

// A sphere class.
class Sphere
{
public:
   float r;        // Radius of sphere
   float x, y, z;  // Coordinates of sphere
   Sphere(float xcoord, float ycoord = 2.0,
          float zcoord = 2.5, float radius = 1.0)
    { x = xcoord; y = ycoord; z = zcoord; r = radius; }
   ~Sphere()
    {
      cout << "Sphere (" << x << ", " << y
           << ", " << z << ", " << r << ") destroyed\n";
    }
    inline float volume()
    {
      return (r * r * r * 4 * PI / 3);
    }
    float surface_area()
    {
     return (r * r * 4 * PI);
    }
};

void main()
{
   Sphere s(1.0);               // use all default
   Sphere t(1.0, 1.1);          // override y coord
   Sphere u(1.0, 1.1, 1.2);     // override y and z
   Sphere v(1.0, 1.1, 1.2, 1.3); // override all defaults
   cout << "s: X = " << s.x << ", Y = " << s.y
        << ", Z = " << s.z << ", R = " << s.r << "\n";
```

```
    cout << "The volume of s is " << s.volume() << "\n";
    cout << "The surface area of s is " << s.surface_area() << "\n";
    cout << "t: X = " << t.x << ", Y = " << t.y
         << ", Z = " << t.z << ", R = " << t.r << "\n";
    cout << "The volume of t is " << t.volume() << "\n";
    cout << "The surface area of t is " << t.surface_area() << "\n";
    cout << "u: X = " << u.x << ", Y = " << u.y
         << ", Z = " << u.z << ", R = " << u.r << "\n";
    cout << "The volume of u is " << u.volume() << "\n";
    cout << "The surface area of u is " << u.surface_area() << "\n";
    cout << "v: X = " << v.x << ", Y = " << v.y
         << ", Z = " << v.z << ", R = " << v.r << "\n";
    cout << "The volume of v is " << v.volume() << "\n";
    cout << "The surface area of v is " << v.surface_area() << "\n";
    return;

}
```

Here is the output from this program:

```
s: X = 1, Y = 2, Z = 2.5, R = 1
The volume of s is 4.188787
The surface area of s is 12.56636
t: X = 1, Y = 1.1, Z = 2.5, R = 1
The volume of t is 4.188787
The surface area of t is 12.56636
u: X = 1, Y = 1.1, Z = 1.2, R = 1
The volume of u is 4.188787
The surface area of u is 12.56636
v: X = 1, Y = 1.1, Z = 1.2, R = 1.3
The volume of v is 9.202764
The surface area of v is 21.237148
Sphere (1, 1.1, 1.2, 1.3) destroyed
Sphere (1, 1.1, 1.2, 1) destroyed
Sphere (1, 1.1, 2.5, 1) destroyed
Sphere (1, 2, 2.5, 1) destroyed
```

Notice that when you use a default value, you must also use the other default values to its right. Similarly, once you define a function's parameter as having a default value, every parameter to its right must have a default value as well.

2. You also can call more than one constructor; this is called *overloading* the constructor. When having more than one constructor, all with the same name of the class, you must give them each a different parameter list so the compiler can determine which one you intend to use. A common use of overloaded constructors is to create an uninitialized object on the receiving end of an assignment, as you see done here:

```
// C32OVCON.CPP
// Demonstrates use of overloaded constructors.

#include  <iostream.h>
#include  <math.h>
const float PI  = 3.14159;  // Approximate value of pi.

// A sphere class.
class Sphere
{
public:
   float r;        // Radius of sphere
   float x, y, z;  // Coordinates of sphere
   Sphere() { /* doesn't do anything... */ }
   Sphere(float xcoord, float ycoord,
          float zcoord, float radius)
   { x = xcoord; y = ycoord; z = zcoord; r = radius; }
   ~Sphere()
   {
      cout << "Sphere (" << x << ", " << y
           << ", " << z << ", " << r << ") destroyed\n";
   }
   inline float volume()
   {
     return (r * r * r * 4 * PI / 3);
   }
   float surface_area()
   {
     return (r * r * 4 * PI);
   }
};

void main()
{
   Sphere s(1.0, 2.0, 3.0, 4.0);
```

```
    Sphere t;    // No parameters (an uninitialized sphere).

    cout << "X = " << s.x << ", Y = " << s.y
         << ", Z = " << s.z << ", R = " << s.r << "\n";
    t = s;
    cout << "The volume of t is " << t.volume() << "\n";
    cout << "The surface area of t is " << t.surface_area()
         << "\n";
    return;
}
```

Class Member Visibility

Recall that the `Sphere()` class had the label `public`. Declaring the `public` label is necessary because, by default, all members of a class are `private`. Private members cannot be accessed by anything but a member function. In order for data or member functions to be used by other programs, they must be explicitly declared `public`. In the case of the `Sphere()` class, you probably want to hide the actual data from other classes. This protects the data's integrity. The next program adds a `cube()` and `square()` function to do some of the work of the `volume()` and `surface_area()` functions. There is no need for other functions to use those member functions.

```
// Filename: C32VISIB.CPP
// Demonstrates use of class visibility labels.

#include  <iostream.h>
#include  <math.h>
const float PI = 3.14159;  // Approximate value of pi.

// A sphere class.
class Sphere
{
private:
   float r;       // Radius of sphere
   float x, y, z;  // Coordinates of sphere
   float cube() { return (r * r * r); }
   float square() { return (r * r); }
```

```
public:
   Sphere(float xcoord, float ycoord, float zcoord, float radius)
   { x = xcoord; y = ycoord; z = zcoord; r = radius; }
   ~Sphere()
   {
     cout << "Sphere (" << x << ", " << y
          << ", " << z << ", " << r << ") destroyed\n";
   }
   float volume()
   {
     return (cube() * 4 * PI / 3);
   }
   float surface_area()
   {
      return (square() * 4 * PI);
   }
};

void main()
{
   Sphere s(1.0, 2.0, 3.0, 4.0);
   cout << "The volume is " << s.volume() << "\n";
   cout << "The surface area is " << s.surface_area() << "\n";
   return;
}
```

Notice that the line showing the data members had to be removed from `main()`. The data members are no longer directly accessible except by a member function of class `Sphere`. In other words, `main()` can never directly manipulate data members such as `r` and `z`, even though it calls the constructor function that created them. The private member data is only visible in the member functions. This is the true power of *data hiding;* even your own code cannot get to the data! The advantage is that you define specific data-retrieval, data-display, and data-changing member functions that `main()` must call to manipulate member data. Through these member functions, you set up a buffer between your program and the program's data structures. If you change the way the data is stored, you do not have to change `main()` or any functions that `main()` calls. You only have to change the member functions of that class.

Review Questions

The answers to the review questions are in Appendix B.

1. What are the two types of class members called?
2. Is a constructor always necessary?
3. Is a destructor always necessary?

4. What is the default visibility of a class data member?
5. How do you make a class member visible outside its class?

Review Exercise

Construct a class to hold personnel records. Use the following data members, and keep them private:

```
char      name[25];
float     salary;
char      date_of_birth[9];
```

Create two constructors, one to initialize the record with its necessary values and another to create an uninitialized record. Create member functions to alter the individual's name, salary, and date of birth.

Summary

You have now been introduced to classes, the data type that distinguishes C++ from its predecessor, C. This was only a cursory glimpse of object-oriented programming. However, you saw that OOP offers an advanced view of your data, combining the data with the member functions that manipulate that data. If you desire to learn more about C++ and become a "guru" of sorts, try *Using Microsoft C/C++ 7* (Que, 0-88022-809-1).