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.

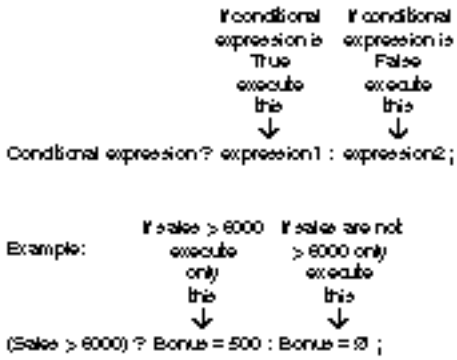


Figure 11.1. Format of the conditional operator.

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);
```

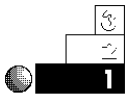
Because each C++ expression has a value—in this case, the value being assigned—this statement could be even more succinct, without loss of readability, by assigning ans the answer to the left of the conditional:

```
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);
```

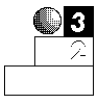
2. Using a conditional operator, you can write a routine to find the minimum value between two variables. This is sometimes called a *minimum routine*. The statement to do this is

```
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.

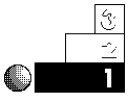
```
x = p + (((amt++)));
```

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:

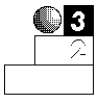
```
sales = ++(rate * hours);  // Not allowed!!
```

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.

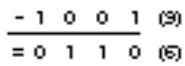


Figure 11.5. Performing bitwise ~ on the number 9.

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.

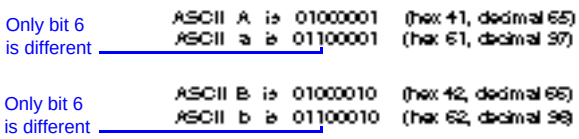


Figure 11.6. Bitwise difference between two uppercase and two lowercase ASCII letters.

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.

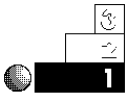
```
// Filename: C11UPCS1.CPP
// Converts the input characters to uppercase
// if they aren't already.
#include <iostream.h>
main()
{
   char first, middle, last;   // Will hold user's initials
   int bitmask=223;                  // 11011111 in binary

   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?

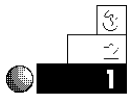
   a. `1 ^ 0 & 1 & 1 | 0`

   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++.