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:

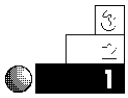
```
if ((sales > old_sales) || (inventory_flag = 'Y'))...
```

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.

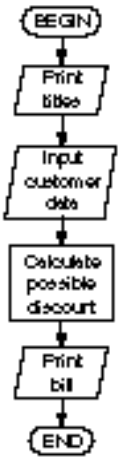
```
// Filename: C10AGE.CPP
// Program that helps ensure age values are reasonable.
#include <iostream.h>
main()
{
   int age;

   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.

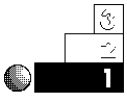
Notice that `||` (OR) has lower precedence than `&&` (AND). Therefore, the following `if` tests are equivalent:

```
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`

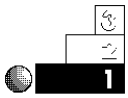
   c. `17 - 1 > 15 + 1 && 0 + 2 != 1 == 1 || 4 != 1`

   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!