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.

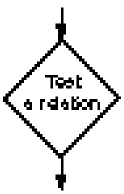
The arrow points to the lesser of the two values. Notice how, in the previous Example 1, the arrow (the point of the `<` or `>`) always points to the lesser number. The larger, open part of the arrow points to the larger number.

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`.

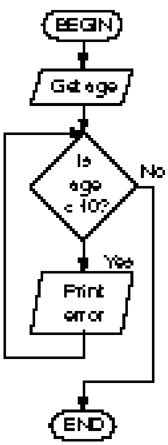
```
Payroll Calculation
-------------------
What is salesperson's last name? Harrison
How many hours did the salesperson work? 40
What were the total sales? 6050.64
Harrison made $164.00
and got a bonus of $0.00
```

```
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 foolproof 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.

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.