




Hüpp c++

CHAPTER 2:  
FLOW OF CONTROL



## Boolean Expressions

*He who would distinguish the true from the false must have an adequate idea of what is true and false.*

Benedict Spinoza, *Ethics*

Most branching statements are controlled by Boolean expressions. A **Boolean expression** is any expression that is either true or false. The simplest form for a Boolean expression consists of two expressions, such as numbers or variables, that are compared with one of the comparison operators shown in Display 2.1. Notice that some of the operators are spelled with two symbols, for example, `==`, `!=`, `<=`, `>=`. Be sure to notice that you use a double equal `==` for the equal sign and that you use the two symbols `!=` for not equal. Such two-symbol operators should not have any space between the two symbols.



## THE “AND” OPERATOR, &&

You can form a more elaborate Boolean expression by combining two simpler Boolean expressions using the “and” operator, &&.

### SYNTAX FOR A BOOLEAN EXPRESSION USING &&

```
(Boolean_Exp_1) && (Boolean_Exp_2)
```

### EXAMPLE (WITHIN AN if-else STATEMENT)

```
if ( (score > 0) && (score < 10) )  
    cout << "score is between 0 and 10.\n";  
else  
    cout << "score is not between 0 and 10.\n";
```

If the value of `score` is greater than 0 and the value of `score` is also less than 10, then the first `cout` statement will be executed; otherwise, the second `cout` statement will be executed. (if-else statements are covered a bit later in this chapter, but the meaning of this simple example should be intuitively clear.)

## Pitfall

### STRINGS OF INEQUALITIES

Do not use a string of inequalities such as `x < z < y`. If you do, your program will probably compile and run, but it will undoubtedly give incorrect output. Instead, you must use two inequalities connected with an &&, as follows:

```
(x < z) && (z < y)
```



## THE “OR” OPERATOR, ||

You can form a more elaborate Boolean expression by combining two simpler Boolean expressions using the “or” operator, ||.

### SYNTAX FOR A BOOLEAN EXPRESSION USING ||

```
(Boolean_Exp_1) || (Boolean_Exp_2)
```

### EXAMPLE WITHIN AN if-else STATEMENT

```
if ( (x == 1) || (x == y) )  
    cout << "x is 1 or x equals y.\n";  
else  
    cout << "x is neither 1 nor equal to y.\n";
```

If the value of *x* is equal to 1 or the value of *x* is equal to the value of *y* (or both), then the first `cout` statement will be executed; otherwise, the second `cout` statement will be executed. (`if-else` statements are covered a bit later in this chapter, but the meaning of this simple example should be intuitively clear.)

You can also combine two comparisons using the “or” operator, which is spelled || in C++. For example, the following is true provided *y* is less than 0 *or* *y* is greater than 12:

```
(y < 0) || (y > 12)
```

When two comparisons are connected using a ||, the entire expression is true provided that one or both of the comparisons are true; otherwise, the entire expression is false.

You can negate any Boolean expression using the ! operator. If you want to negate a Boolean expression, place the expression in parentheses and place the ! operator in front of it. For example, !(*x* < *y*) means “*x* is *not* less than *y*.” The ! operator can usually be avoided. For example, !(*x* < *y*) is equivalent to *x* >= *y*. In some cases you can safely omit the parentheses, but the parentheses never do any harm. The exact details on omitting parentheses are given in the subsection entitled **Precedence Rules**.



MATH SYMBOL	ENGLISH	C++ NOTATION	C++ SAMPLE	MATH EQUIVALENT
=	Equal to	==	<code>x + 7 == 2*y</code>	$x + 7 = 2y$
≠	Not equal to	!=	<code>ans != 'n'</code>	$ans \neq 'n'$
<	Less than	<	<code>count &lt; m + 3</code>	$count < m + 3$
≤	Less than or equal to	<=	<code>time &lt;= limit</code>	$time \leq limit$
>	Greater than	>	<code>time &gt; limit</code>	$time > limit$
≥	Greater than or equal to	>=	<code>age &gt;= 21</code>	$age \geq 21$



## AND

<i>Exp_1</i>	<i>Exp_2</i>	<i>Exp_1 &amp;&amp; Exp_2</i>
true	true	true
true	false	false
false	true	false
false	false	false

## OR

<i>Exp_1</i>	<i>Exp_2</i>	<i>Exp_1    Exp_2</i>
true	true	true
true	false	true
false	true	true
false	false	false

## NOT

<i>Exp</i>	<i>!(Exp)</i>
true	false
false	true

### THE BOOLEAN (bool) VALUES ARE true AND false

true and false are predefined constants of type bool. (They must be written in lowercase.) In C++, a Boolean expression evaluates to the bool value true when it is satisfied and to the bool value false when it is not satisfied.



## Display 2.3 Precedence of Operators (part 1 of 2)

<code>::</code>	Scope resolution operator
<code>.</code> <code>-&gt;</code> <code>[]</code> <code>()</code> <code>++</code> <code>--</code>	Dot operator Member selection Array indexing Function call Postfix increment operator (placed after the variable) Postfix decrement operator (placed after the variable)
<code>++</code> <code>--</code> <code>!</code> <code>-</code> <code>+</code> <code>*</code> <code>&amp;</code> <code>new</code> <code>delete</code> <code>delete[]</code> <code>sizeof</code> <code>()</code>	Prefix increment operator (placed before the variable) Prefix decrement operator (placed before the variable) Not Unary minus Unary plus Dereference Address of Create (allocate memory) Destroy (deallocate) Destroy array (deallocate) Size of object Type cast
<code>*</code> <code>/</code> <code>%</code>	Multiply Divide Remainder (modulo)
<code>+</code> <code>-</code>	Addition Subtraction
<code>&lt;&lt;</code> <code>&gt;&gt;</code>	Insertion operator (console output) Extraction operator (console input)

*Highest precedence  
(done first)*




*Lower precedence  
(done later)*



### Display 2.3 Precedence of Operators (part 2 of 2)

All operators in part 2 are of lower precedence than those in part 1.

<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to
==	Equal
!=	Not equal
&&	And
	Or
=	Assignment
+=	Add and assign
-=	Subtract and assign
*=	Multiply and assign
/=	Divide and assign
%=	Modulo and assign
? :	Conditional operator
throw	Throw an exception
,	Comma operator



Lowest precedence  
(done last)





The previous description of how a Boolean expression is evaluated is basically correct, but in C++, the computer actually takes an occasional shortcut when evaluating a Boolean expression. Notice that in many cases you need to evaluate only the first of two subexpressions in a Boolean expression. For example, consider the following:

```
(x >= 0) && (y > 1)
```

If  $x$  is negative, then  $(x \geq 0)$  is *false*. As you can see in the tables in Display 2.1, when one subexpression in an `&&` expression is *false*, then the whole expression is *false*, no matter whether the other expression is *true* or *false*. Thus, if we know that the first expression is *false*, there is no need to evaluate the second expression. A similar thing happens with `||` expressions. If the first of two expressions joined with the `||` operator is *true*, then you know the entire expression is *true*, no matter whether the second expression is *true* or *false*. The C++ language uses this fact to sometimes save itself the trouble of evaluating the second subexpression in a logical expression connected with an `&&` or `||`. C++ first evaluates the leftmost of the two expressions joined by an `&&` or `||`. If that gives it enough information to determine the final value of the expression (independent of the value of the second expression), then C++ does not bother to evaluate the second expression. This method of evaluation is called **short-circuit evaluation**.

short-circuit  
evaluation



Some languages other than C++ use **complete evaluation**. In complete evaluation, when two expressions are joined by an `&&` or `||`, both subexpressions are always evaluated and then the truth tables are used to obtain the value of the final expression. complete evaluation

Both short-circuit evaluation and complete evaluation give the same answer, so why should you care that C++ uses short-circuit evaluation? Most of the time you need not care. As long as both subexpressions joined by the `&&` or the `||` have a value, the two methods yield the same result. However, if the second subexpression is undefined, you might be happy to know that C++ uses short-circuit evaluation. Let's look at an example that illustrates this point. Consider the following statement:

```
if ( (kids != 0) && ((pieces/kids) >= 2) )  
    cout << "Each child may have two pieces!";
```

If the value of `kids` is not zero, this statement involves no subtleties. However, suppose the value of `kids` is zero; consider how short-circuit evaluation handles this case. The expression `(kids != 0)` evaluates to `false`, so there would be no need to evaluate the

second expression. Using short-circuit evaluation, C++ says that the entire expression is `false`, without bothering to evaluate the second expression. This prevents a run-time error, since evaluating the second expression would involve dividing by zero.



## ■ if-else STATEMENTS

An **if-else statement** chooses between two alternative statements based on the value of a Boolean expression. For example, suppose you want to design a program to compute a week's salary for an hourly employee. Assume the firm pays an overtime rate of one-and-one-half times the regular rate for all hours after the first 40 hours worked. When the employee works 40 or more hours, the pay is then equal to

$$\text{rate} \times 40 + 1.5 \times \text{rate} \times (\text{hours} - 40)$$

However, if the employee works less than 40 hours, the correct pay formula is simply

$$\text{rate} \times \text{hours}$$

The following **if-else** statement computes the correct pay for an employee whether the employee works less than 40 hours or works 40 or more hours,

```
if (hours > 40)
    grossPay = rate*40 + 1.5*rate*(hours - 40);
else
    grossPay = rate*hours;
```

The syntax for an **if-else** statement is given in the accompanying box. If the Boolean expression in parentheses (after the **if**) evaluates to **true**, then the statement before the **else** is executed. If the Boolean expression evaluates to **false**, the statement after the **else** is executed.



## SYNTAX: A SEQUENCE OF STATEMENTS FOR EACH ALTERNATIVE

```
if (Boolean_Expression)
{
    Yes_Statement_1
    Yes_Statement_2
    ...
    Yes_Statement_Last
}
else
{
    No_Statement_1
    No_Statement_2
    ...
    No_Statement_Last
}
```

### EXAMPLE

```
if (myScore > yourScore)
{
    cout << "I win!\n";
    wager = wager + 100;
}
else
{
    cout << "I wish these were golf scores.\n";
    wager = 0;
}
```



## ■ COMPOUND STATEMENTS

You will often want the branches of an `if-else` statement to execute more than one statement each. To accomplish this, enclose the statements for each branch between a pair of braces, `{` and `}`, as indicated in the second syntax template in the box entitled **if-else Statement**. A list of statements enclosed in a pair of braces is called a **compound statement**. A compound statement is treated as a single statement by C++ and may be used anywhere that a single statement may be used. (Thus, the second syntax template in the box entitled `if-else Statement`. is really just a special case of the first one.)

There are two commonly used ways of indenting and placing braces in `if-else` statements, which are illustrated below:

```
if (myScore > yourScore)
{
    cout << "I win!\n";
    wager = wager + 100;
}
else
{
    cout << "I wish these were golf scores.\n";
    wager = 0;
}
```

and

```
if (myScore > yourScore){
    cout << "I win!\n";
    wager = wager + 100;
} else {
    cout << "I wish these were golf scores.\n";
    wager = 0;
}
```

The only differences are the placement of braces. We find the first form easier to read and therefore prefer it. The second form saves lines, and so some programmers prefer the second form or some minor variant of it.



## OMITTING THE `else`

Sometimes you want one of the two alternatives in an `if-else` statement to do nothing at all. In C++ this can be accomplished by omitting the `else` part. These sorts of statements are referred to as **if statements** to distinguish them from `if-else` statements. For example, the first of the following two statements is an `if` statement:

```
if (sales >= minimum)
    salary = salary + bonus;
cout << "salary = $" << salary;
```

If the value of `sales` is greater than or equal to the value of `minimum`, the assignment statement is executed and then the following `cout` statement is executed. On the other hand, if the value of `sales` is less than `minimum`, then the embedded assignment statement is not executed. Thus, the `if` statement causes no change (that is, no bonus is added to the base salary), and the program proceeds directly to the `cout` statement.





## MULTIWAY if-else STATEMENT

### SYNTAX

```
if (Boolean_Expression_1)
    Statement_1
else if (Boolean_Expression_2)
    Statement_2
    .
    .
    .
else if (Boolean_Expression_n)
    Statement_n
else
    Statement_For_All_Other_Possibilities
```

### EXAMPLE

```
if ((temperature < -10) && (day == SUNDAY))
    cout << "Stay home.";
else if (temperature < -10) //and day != SUNDAY
    cout << "Stay home, but call work.";
else if (temperature <= 0) //and temperature >= -10
    cout << "Dress warm.";
else //temperature > 0
    cout << "Work hard and play hard.";
```

The Boolean expressions are checked in order until the first true Boolean expression is encountered, and then the corresponding statement is executed. If none of the Boolean expressions is true, then the *Statement\_For\_All\_Other\_Possibilities* is executed.



## switch STATEMENT

### SYNTAX

```
switch (Controlling_Expression)
{
    case Constant_1:
        Statement_Sequence_1
        break;
    case Constant_2:
        Statement_Sequence_2
        break;
        .
        .
        .
    case Constant_n:
        Statement_Sequence_n
        break;
    default:
        Default_Statement_Sequence
}
```

*You need not place a break statement in each case. If you omit a break, that case continues until a break (or the end of the switch statement) is reached.*

### EXAMPLE

```
int vehicleClass;
double toll;
cout << "Enter vehicle class: ";
cin >> vehicleClass;

switch (vehicleClass)
{
    case 1:
        cout << "Passenger car.";
        toll = 0.50;
        break;
    case 2:
        cout << "Bus.";
        toll = 1.50;
        break;
    case 3:
        cout << "Truck.";
        toll = 2.00;
        break;
    default:
        cout << "Unknown vehicle class!";
}
```

*If you forget this break, then passenger cars will pay \$1.50.*





Note that you can have two case labels for the same section of code, as in the following portion of a switch statement:

```
case 'A':  
case 'a':  
    cout << "Excellent. "  
        << "You need not take the final.\n";  
break;
```

Since the first case has no break statement (in fact, no statement at all), the effect is the same as having two labels for one case, but C++ syntax requires one keyword case for each label, such as 'A' and 'a'.

If no case label has a constant that matches the value of the controlling expression, then the statements following the default label are executed. You need not have a default section. If there is no default section and no match is found for the value of the controlling expression, then nothing happens when the switch statement is executed. However, it is safest to always have a default section. If you think your case labels list all possible outcomes, then you can put an error message in the default section.

default

## Pitfall

### FORGETTING A break IN A switch STATEMENT

If you forget a break in a switch statement, the compiler will not issue an error message. You will have written a syntactically correct switch statement, but it will not do what you intended it to do. Notice the annotation in the example in the box entitled **switch Statement**.

## Tip

### USE switch STATEMENTS FOR MENUS

The multiway if-else statement is more versatile than the switch statement, and you can use a multiway if-else statement anywhere you can use a switch statement. However, sometimes the switch statement is clearer. For example, the switch statement is perfect for implementing menus. Each branch of the switch statement can be one menu choice.



## ■ ENUMERATION TYPES

An **enumeration type** is a type whose values are defined by a list of constants of type `int`. An enumeration type is very much like a list of declared constants. Enumeration types can be handy for defining a list of identifiers to use as the case labels in a `switch` statement.

When defining an enumeration type, you can use any `int` values and can define any number of constants. For example, the following enumeration type defines a constant for the length of each month:

```
enum MonthLength { JAN_LENGTH = 31, FEB_LENGTH = 28,
    MAR_LENGTH = 31, APR_LENGTH = 30, MAY_LENGTH = 31,
    JUN_LENGTH = 30, JUL_LENGTH = 31, AUG_LENGTH = 31,
    SEP_LENGTH = 30, OCT_LENGTH = 31, NOV_LENGTH = 30,
    DEC_LENGTH = 31 };
```

As this example shows, two or more named constants in an enumeration type can receive the same `int` value.

If you do not specify any numeric values, the identifiers in an enumeration type definition are assigned consecutive values beginning with `0`. For example, the type definition

```
enum Direction { NORTH = 0, SOUTH = 1, EAST = 2, WEST = 3 };
```

is equivalent to

```
enum Direction { NORTH, SOUTH, EAST, WEST };
```

The form that does not explicitly list the `int` values is normally used when you just want a list of names and do not care about what values they have.

Suppose you initialize an enumeration constant to some value, say

```
enum MyEnum { ONE = 17, TWO, THREE, FOUR = -3, FIVE };
```

then `ONE` takes the value 17; `TWO` takes the next `int` value, 18; `THREE` takes the next value, 19; `FOUR` takes `-3`; and `FIVE` takes the next value, `-2`. In short, the default for the first enumeration constant is `0`. The rest increase by 1 unless you set one or more of the enumeration constants.



## ■ THE CONDITIONAL OPERATOR

It is possible to embed a conditional inside an expression by using a ternary operator known as the **conditional operator** (also called the *ternary operator* or *arithmetic if*). Its use is reminiscent of an older programming style, and we do not advise using it. It is included here for the sake of completeness (and in case you disagree with our programming style).

The conditional operator is a notational variant on certain forms of the `if-else` statement. This variant is illustrated below. Consider the statement

```
if (n1 > n2)
    max = n1;
else
    max = n2;
```

This can be expressed using the conditional operator as follows:

```
max = (n1 > n2) ? n1 : n2;
```

The expression on the right-hand side of the assignment statement is the **conditional operator expression**:

```
(n1 > n2) ? n1 : n2
```

conditional  
operator  
expression

The `?` and `:` together form a ternary operator known as the conditional operator. A conditional operator expression starts with a Boolean expression followed by a `?` and then followed by two expressions separated with a colon. If the Boolean expression is `true`, then the first of the two expressions is returned; otherwise, the second of the two expressions is returned.





## ■ THE while AND do-while STATEMENTS

### SYNTAX FOR while AND do-while STATEMENTS

#### A while STATEMENT WITH A SINGLE STATEMENT BODY

```
while (Boolean_Expression)  
    Statement
```

#### A while STATEMENT WITH A MULTISTatement BODY

```
while (Boolean_Expression)  
{  
    Statement_1  
    Statement_2  
    .  
    .  
    .  
    Statement_Last  
}
```

#### A do-while STATEMENT WITH A SINGLE-STATEMENT BODY

```
do  
    Statement  
while (Boolean_Expression);
```

#### A do-while STATEMENT WITH A MULTISTatement BODY

```
do  
{  
    Statement_1  
    Statement_2  
    .  
    .  
    .  
    Statement_Last  
} while (Boolean_Expression);
```

*Do not forget the  
final semicolon.*



## Example of a while Statement (part 1 of 2)

---

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

3  int main( )
4  {
5      int countDown;

6      cout << "How many greetings do you want? ";
7      cin >> countDown;

8      while (countDown > 0)
9      {
10         cout << "Hello ";
11         countDown = countDown - 1;
12     }

13     cout << endl;
14     cout << "That's all!\n";

15     return 0;
16 }
```

---

### SAMPLE DIALOGUE 1

```
How many greetings do you want? 3
Hello Hello Hello
That's all!
```

### SAMPLE DIALOGUE 2

```
How many greetings do you want? 0
That's all!
```

*The loop body is executed  
zero times*



### Example of a do-while Statement (part 1 of 2)

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

3  int main( )
4  {
5      int countDown;

6      cout << "How many greetings do you want? ";
7      cin >> countDown;

8      do
9      {
10         cout << "Hello ";
11         countDown = countDown - 1;
12     }while (countDown > 0);

13     cout << endl;
14     cout << "That's all!\n";

15     return 0;
16 }
```

#### SAMPLE DIALOGUE 1

```
How many greetings do you want? 3
Hello Hello Hello
That's all!
```

#### SAMPLE DIALOGUE 2

```
How many greetings do you want? 0
Hello
That's all!
```

*The loop body  
is always executed at least once.*



# INCREMENT AND DECREMENT OPERATORS REVISITED

## The Increment Operator in an Expression

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

3  int main( )
4  {
5      int numberOfItems, count,
6          caloriesForItem, totalCalories;

7      cout << "How many items did you eat today? ";
8      cin >> numberOfItems;

9      totalCalories = 0;
10     count = 1;
11     cout << "Enter the number of calories in each of the\n"
12          << numberOfItems << " items eaten:\n";

13     while (count++ <= numberOfItems)
14     {
15         cin >> caloriesForItem;
16         totalCalories = totalCalories
17             + caloriesForItem;
18     }

19     cout << "Total calories eaten today = "
20          << totalCalories << endl;
21     return 0;
22 }
```

### SAMPLE DIALOGUE

```
How many items did you eat today? 7
Enter the number of calories in each of the
7 items eaten:
300 60 1200 600 150 1 120
Total calories eaten today = 2431
```



## ■ THE COMMA OPERATOR

The **comma operator** is a way of evaluating a list of expressions and returning the value of the last expression. It is sometimes handy to use in a `for` loop, as indicated in our discussion of the `for` loop in the next subsection. We do not advise using it in other contexts, but it is legal to use it in any expression.

The comma operator is illustrated by the following assignment statement:

```
result = (first = 2, second = first + 1);
```

The comma operator is the comma shown. The **comma expression** is the expression on the right-hand side of the assignment operator. The comma operator has two expressions as operands. In this case the two operands are

```
first = 2 and second = first + 1
```

The first expression is evaluated, and then the second expression is evaluated. As you may recall from Chapter 1, the assignment statement when used as an expression returns the new value of the variable on the left side of the assignment operator. So, this comma expression returns the final value of the variable `second`, which means that the variable `result` is set equal to 3.

Since only the value of the second expression is returned, the first expression is evaluated solely for its side effects. In the above example, the side effect of the first expression is to change the value of the variable `first`.

You may have a longer list of expressions connected with commas, but you should only do so when the order of evaluation is not important. If the order of evaluation is important, you should use parentheses. For example:

```
result = ((first = 2, second = first + 1), third = second + 1);
```

sets the value of `result` equal to 4. However, the value that the following gives to `result` is unpredictable, because it does not guarantee that the expressions are evaluated in order:

```
result = (first = 2, second = first + 1, third = second + 1);
```

For example, `third = second + 1` might be evaluated before `second = first + 1`.<sup>1</sup>





## THE for STATEMENT

The third and final loop statement in C++ is the **for statement**. The for statement is most commonly used to step through some integer variable in equal increments. As we will see in Chapter 5, the for statement is often used to step through an array. The for statement is, however, a completely general looping mechanism that can do anything that a while loop can do.

For example, the following for statement sums the integers 1 through 10:

```
sum = 0;
for (n = 1; n <= 10; n++)
    sum = sum + n;
```

A for statement begins with the keyword `for` followed by three things in parentheses that tell the computer what to do with the controlling variable. The beginning of a for statement looks like the following:

```
for (Initialization_Action; Boolean_Expression; Update_Action)
```

The first expression tells how the variable, variables, or other things are initialized; the second gives a Boolean expression that is used to check for when the loop should end; and the last expression tells how the loop control variable is updated after each iteration of the loop body.

```
for (number = 100; number >= 0; number--)
{
    cout << number
        << " bottles of beer on the shelf.\n";
    if (number > 0)
        cout << "Take one down and pass it around.\n";
}
```



## FOR STATEMENT SYNTAX

```
for (Initialization_Action; Boolean_Expression; Update_Action)
    Body_Statement
```

### EXAMPLE

```
for (number = 100; number >= 0; number--)
    cout << number
        << " bottles of beer on the shelf.\n";
```

## EQUIVALENT while LOOP SYNTAX

```
Initialization_Action;
while (Boolean_Expression)
{
    Body_Statement
    Update_Action;
}
```

### EQUIVALENT EXAMPLE

```
number = 100;
while (number >= 0)
{
    cout << number
        << " bottles of beer on the shelf.\n";
    number--;
}
```

### SAMPLE DIALOGUE

```
100 bottles of beer on the shelf.
99 bottles of beer on the shelf.
.
.
.
0 bottles of beer on the shelf.
```

## Tip

### REPEAT-N-TIMES LOOPS

A for statement can be used to produce a loop that repeats the loop body a predetermined number of times. For example, the following is a loop body that repeats its loop body three times:

```
for (int count = 1; count <= 3; count++)
    cout << "Hip, Hip, Hurray\n";
```

The body of a for statement need not make any reference to a loop control variable, such as the variable count.



## EXTRA SEMICOLON IN A FOR STATEMENT

You normally do not place a semicolon after the parentheses at the beginning of a for loop. To see what can happen, consider the following for loop:

```
for (int count = 1; count <= 10; count++); ← Problem semicolon
    cout << "Hello\n";
```

If you did not notice the extra semicolon, you might expect this for loop to write Hello to the screen ten times. If you do notice the semicolon, you might expect the compiler to issue an error message. Neither of those things happens. If you embed this for loop in a complete program, the compiler will not complain. If you run the program, only one Hello will be output instead of ten Hellos. What is happening? To answer that question, we need a little background.

One way to create a statement in C++ is to put a semicolon after something. If you put a semicolon after `x++`, you change the expression

```
x++
```

into the statement

```
x++;
```

If you place a semicolon after nothing, you still create a statement. Thus, the semicolon by itself is a statement, which is called the **empty statement** or the **null statement**. The empty statement performs no action, but it still is a statement. Therefore, the following is a complete and legitimate for loop, whose body is the empty statement:

```
for (int count = 1; count <= 10; count++);
```

This for loop is indeed iterated ten times, but since the body is the empty statement, nothing happens when the body is iterated. This loop does nothing, and it does nothing ten times!

This same sort of problem can arise with a while loop. Be careful not to place a semicolon after the closing parenthesis that encloses the Boolean expression at the start of a while loop. A do-while loop has just the opposite problem. You must remember always to end a do-while loop with a semicolon.



## INFINITE LOOPS

A `while` loop, `do-while` loop, or `for` loop does not terminate as long as the controlling Boolean expression is true. This Boolean expression normally contains a variable that will be changed by the loop body, and usually the value of this variable is changed in a way that eventually makes the Boolean expression false and therefore terminates the loop. However, if you make a mistake and write your program so that the Boolean expression is always true, then the loop will run forever. A loop that runs forever is called an **infinite loop**.

Unfortunately, examples of infinite loops are not hard to come by. First let's describe a loop that does terminate. The following C++ code will write out the positive even numbers less than 12. That is, it will output the numbers 2, 4, 6, 8, and 10, one per line, and then the loop will end.

```
x = 2;
while (x != 12)
{
    cout << x << endl;
    x = x + 2;
}
```

The value of `x` is increased by 2 on each loop iteration until it reaches 12. At that point, the Boolean expression after the word `while` is no longer true, so the loop ends.

Now suppose you want to write out the odd numbers less than 12, rather than the even numbers. You might mistakenly think that all you need do is change the initializing statement to

```
x = 1;
```

But this mistake will create an infinite loop. Because the value of `x` goes from 11 to 13, the value of `x` is never equal to 12; thus, the loop will never terminate.

This sort of problem is common when loops are terminated by checking a numeric quantity using `==` or `!=`. When dealing with numbers, it is always safer to test for passing a value. For example, the following will work fine as the first line of our `while` loop:

```
while (x < 12)
```

With this change, `x` can be initialized to any number and the loop will still terminate.

A program that is in an infinite loop will run forever unless some external force stops it. Since you can now write programs that contain an infinite loop, it is a good idea to learn how to force a program to terminate. The method for forcing a program to stop varies from system to system. The keystrokes Control-C will terminate a program on many systems. (To type Control-C, hold down the Control key while pressing the C key.)

In simple programs, an infinite loop is almost always an error. However, some programs are intentionally written to run forever (in principle), such as the main outer loop in an airline reservation program, which just keeps asking for more reservations until you shut down the computer (or otherwise terminate the program in an atypical way).





# THE break AND continue STATEMENTS

## A break Statement in a Loop

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

3  int main( )
4  {
5      int number, sum = 0, count = 0;
6      cout << "Enter 4 negative numbers:\n";

7      while (++count <= 4)
8      {
9          cin >> number;

10         if (number >= 0)
11         {
12             cout << "ERROR: positive number"
13                 << " or zero was entered as the\n"
14                 << count << "th number! Input ends "
15                 << "with the " << count << "th number.\n"
16                 << count << "th number was not added in.\n";
17             break;
18         }

19         sum = sum + number;
20     }

21     cout << sum << " is the sum of the first "
22         << (count - 1) << " numbers.\n";

23     return 0;
24 }
```

## SAMPLE DIALOGUE

```
Enter 4 negative numbers:
-1 -2 3 -4
ERROR: positive number or zero was entered as the
3rd number! Input ends with the 3rd number.
3rd number was not added in
-3 is the sum of the first 2 numbers.
```



## A continue Statement in a Loop

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

3  int main( )
4  {
5      int number, sum = 0, count = 0;
6      cout << "Enter 4 negative numbers, ONE PER LINE:\n";

7      while (count < 4)
8      {
9          cin >> number;

10         if (number >= 0)
11         {
12             cout << "ERROR: positive number (or zero)!\n"
13                 << "Reenter that number and continue:\n";
14             continue;
15         }

16         sum = sum + number;
17         count++;
18     }

19     cout << sum << " is the sum of the "
20         << count << " numbers.\n";
21     return 0;
22 }
```

### SAMPLE DIALOGUE

```
Enter 4 negative numbers, ONE PER LINE:
1
ERROR: positive number (or zero)!
Reenter that number and continue:
-1
-2
3
ERROR: positive number!
Reenter that number and continue:
-3
```

