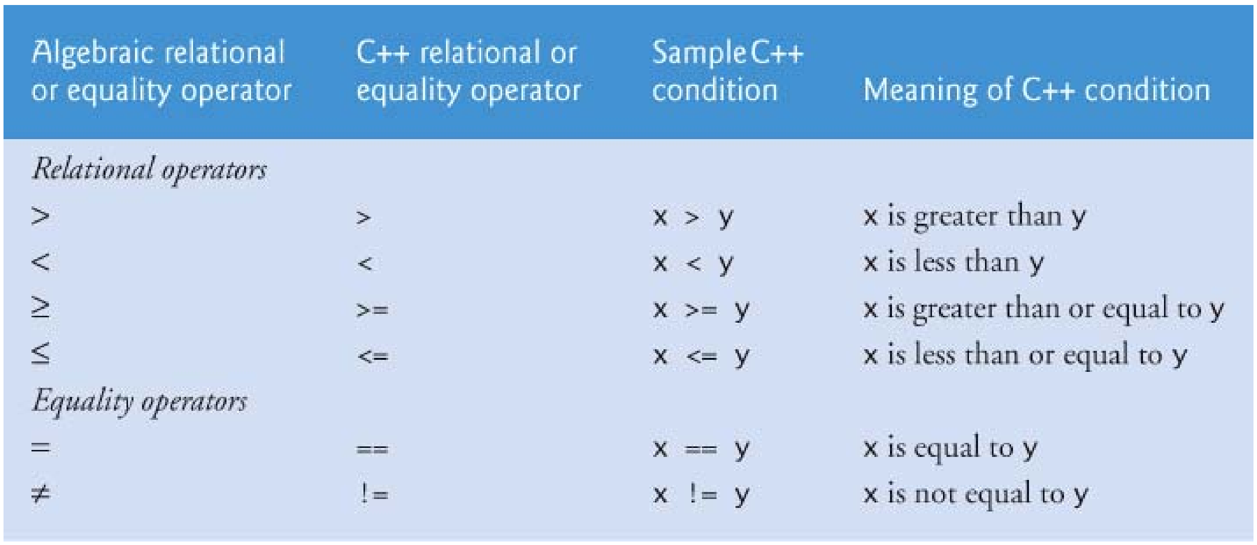
2.7. Decision Making: Equality and Relational Operators

We now introduce a simple version of C++'s **`if` statement** that allows a program to take alternative action based on whether a **condition** is true or false. If the condition is *true*, the statement in the body of the `if` statement *is* executed. If the condition is *false*, the body statement *is not* executed. We'll see an example shortly.

Conditions in `if` statements can be formed by using the **relational operators** and **equality operators** summarized in Fig. 2.12. The relational operators all have the same level of precedence and associate left to right. The equality operators both have the same level of precedence, which is *lower* than that of the relational operators, and associate left to right.



Algebraic relational or equality operator	C++ relational or equality operator	Sample C++ condition	Meaning of C++ condition
Relational operators			
$>$	`>`	`x > y`	x is greater than y
$<$	`<`	`x < y`	x is less than y
$\geq$	`>=`	`x >= y`	x is greater than or equal to y
$\leq$	`<=`	`x <= y`	x is less than or equal to y
Equality operators			
$=$	`==`	`x == y`	x is equal to y
$\neq$	`!=`	`x != y`	x is not equal to y

Fig. 2.12. Relational and equality operators.

Common Programming Error 2.3

*Reversing the order of the pair of symbols in the operators `!=`, `>=` and `<=` (by writing them as `=!`, `=>` and `=<`, respectively) is normally a syntax error. In some cases, writing `!=` as `=!` will not be a syntax error, but almost certainly will be a **logic error** that has an effect at execution time. You'll understand why when you learn about logical operators in Chapter 5. A **fatal logic error** causes a program to fail and terminate prematurely. A **nonfatal logic error** allows a program to continue executing, but usually produces incorrect results.*

Common Programming Error 2.4

Confusing the equality operator `==` with the assignment operator `=` results in logic errors. We like to read the equality operator as "is equal to" or "double equals," and the assignment operator as "gets" or "gets the value of" or "is assigned the value of." As you'll see in Section 5.9, confusing these operators may not necessarily cause an easy-to-recognize syntax error, but may cause subtle logic errors.

Using the `if` Statement

The following example (Fig. 2.13) uses six `if` statements to compare two numbers input by the user. If the condition in any of these `if` statements is satisfied, the output statement associated with that `if` statement is executed.

Click here to view code image

```
// Fig. 2.13: fig02_13.cpp
// Comparing integers using if statements, relational operators
// and equality operators.
#include <iostream> // allows program to perform input and output

using std::cout; // program uses cout
using std::cin; // program uses cin
using std::endl; // program uses endl

// function main begins program execution
int main()
{
   int number1 = 0; // first integer to compare (initialized to 0)
   int number2 = 0; // second integer to compare (initialized to 0)

   cout << "Enter two integers to compare: "; // prompt user for data
   cin >> number1 >> number2; // read two integers from user

   if ( number1 == number2 )
      cout << number1 << " == " << number2 << endl;

   if ( number1 != number2 )
      cout << number1 << " != " << number2 << endl;

   if ( number1 < number2 )
      cout << number1 << " < " << number2 << endl;

   if ( number1 > number2 )
      cout << number1 << " > " << number2 << endl;

   if ( number1 <= number2 )
      cout << number1 << " <= " << number2 << endl;

   if ( number1 >= number2 )
      cout << number1 << " >= " << number2 << endl;
} // end function main
```

```
Enter two integers to compare: 3 7
3 != 7
3 < 7
3 <= 7
```

```
Enter two integers to compare: 22 12
22 != 12
22 > 12
22 >= 12
```

```
Enter two integers to compare: 7 7
7 == 7
7 <= 7
7 >= 7
```

Fig. 2.13. Comparing integers using `if` statements, relational operators and equality operators.

`using` Declarations

Lines 6–8

Click here to view code image

```
using std::cout; // program uses cout
using std::cin; // program uses cin
using std::endl; // program uses endl
```

are **`using` declarations** that eliminate the need to repeat the `std::` prefix as we did in earlier programs. We can now write `cout` instead of `std::cout` , `cin` instead of `std::cin` and `endl` instead of `std::endl` , respectively, in the remainder of the program.

In place of lines 6–8, many programmers prefer to provide the **`using` directive**

```
using namespace std;
```

which enables a program to use *all* the names in any standard C++ header (such as `<iostream>`) that a program might include. From this point forward in the book, we'll use the preceding directive in our programs.[1]

[1]. In Chapter 23, Other Topics, we'll discuss some issues with `using` directives in large-scale systems.

Variable Declarations and Reading the Inputs from the User

Lines 13–14

Click here to view code image

```
int number1 = 0; // first integer to compare (initialized to 0)
int number2 = 0; // second integer to compare (initialized to 0)
```

declare the variables used in the program and initializes them to `0` .

The program uses cascaded stream extraction operations (line 17) to input two integers. Remember that we're allowed to write `cin` (instead of `std::cin`) because of line 7. First a value is read into variable `number1` , then a value is read into variable `number2` .

Comparing Numbers

The `if` statement in lines 19–20

Click here to view code image

```
if ( number1 == number2 )
   cout << number1 << " == " << number2 << endl;
```

compares the values of variables `number1` and `number2` to test for equality. If the values are equal, the statement in line 20 displays a line of text indicating that the numbers are equal. If the conditions are `true` in one or more of the `if` statements starting in lines 22, 25, 28, 31 and 34, the corresponding body statement displays an appropriate line of text.

Each `if` statement in Fig. 2.13 has a single statement in its body and each body statement is indented. In Chapter 4 we show how to specify `if` statements with multiple-statement bodies (by enclosing the body statements in a pair of braces, `{}` , creating what's called a **compound statement** or a **block**).

Good Programming Practice 2.9

Indent the statement(s) in the body of an `if` statement to enhance readability.

Common Programming Error 2.5

Placing a semicolon immediately after the right parenthesis after the condition in an `if` statement is often a logic error (although not a syntax error). The semicolon causes the body of the `if` statement to be empty, so the `if` statement performs no action, regardless of whether or not its condition is true. Worse yet, the original body statement of the `if` statement now becomes a statement in sequence *with the `if` statement and always executes, often causing the program to produce incorrect results.*

White Space

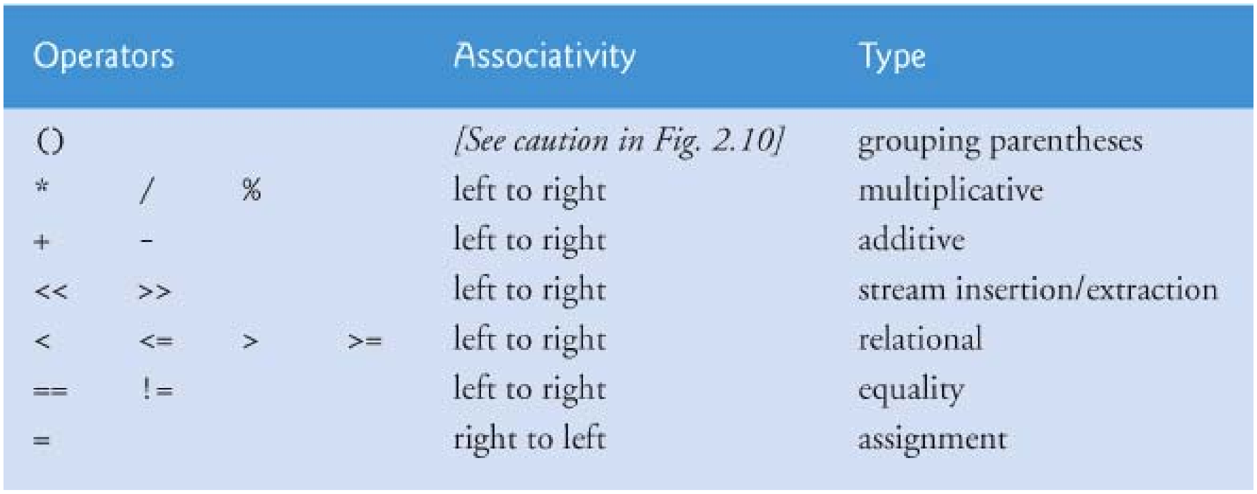
Note the use of white space in Fig. 2.13. Recall that white-space characters, such as tabs, newlines and spaces, are normally ignored by the compiler. So, statements may be split over several lines and may be spaced according to your preferences. It's a syntax error to split identifiers, strings (such as `"hello"`) and constants (such as the number `1000`) over several lines.

Good Programming Practice 2.10

A lengthy statement may be spread over several lines. If a single statement must be split across lines, choose meaningful breaking points, such as after a comma in a comma-separated list, or after an operator in a lengthy expression. If a statement is split across two or more lines, indent all subsequent lines and left-align the group of indented lines.

Operator Precedence

Figure 2.14 shows the precedence and associativity of the operators introduced in this chapter. The operators are shown top to bottom in decreasing order of precedence. All these operators, with the exception of the assignment operator `=`, associate from left to right. Addition is left-associative, so an expression like `x + y + z` is evaluated as if it had been written `(x + y)+ z`. The assignment operator `=` associates from *right to left*, so an expression such as `x = y = 0` is evaluated as if it had been written `x = (y = 0)`, which, as we'll soon see, first assigns `0` to `y`, then assigns the *result* of that assignment—`0`—to `x`.



Operators				Associativity	Type
()				*[See caution in Fig. 2.10]*	grouping parentheses
*	/	%		left to right	multiplicative
+	-			left to right	additive
<<	>>			left to right	stream insertion/extraction
<	<=	>	>=	left to right	relational
==	!=			left to right	equality
=				right to left	assignment

Fig. 2.14. Precedence and associativity of the operators discussed so far.

Good Programming Practice 2.11

Refer to the operator precedence and associativity chart (Appendix A) when writing expressions containing many operators. Confirm that the operators in the expression are performed in the order you expect. If you're uncertain about the order of evaluation in a complex expression, break the expression into smaller statements or use parentheses to force the order of evaluation, exactly as you'd do in an algebraic expression. Be sure to observe that some operators such as assignment (=) associate right to left rather than left to right.