# Copperbelt University
# School of Information and Communication Technology
# Computer Science Department

## Introduction to Programming in C++

***Extracted from Sam's Teach yourself C++ in 21 Days***

### Enumerated Constants

Enumerated constants enable you to create new types and then to define variables of those types whose values are restricted to a set of possible values. For example, you can declare `COLOR` to be an enumeration, and you can define that there are five values for `COLOR`: `RED`, `BLUE`, `GREEN`, `WHITE`, and `BLACK`. The syntax for enumerated constants is to write the keyword *`enum`*, followed by the type name, an open curly brace, each of the legal values separated by a comma, and finally a closing curly brace and a semicolon. Here's an example:

```
enum COLOR { RED, BLUE, GREEN, WHITE, BLACK };
```

This statement performs two tasks:

1. It makes `COLOR` the name of an enumeration, that is, a new type.
2. It makes `RED` a symbolic constant with the value `0`, `BLUE` a symbolic constant with the value `1`, `GREEN` a symbolic constant with the value `2`, and so forth.

Every enumerated constant has an integer value. If you don't specify otherwise, the first constant will have the value `0`, and the rest will count up from there. Any one of the constants can be initialized with a particular value, however, and those that are not initialized will count upward from the ones before them. Thus, if you write

```
enum Color { RED=100, BLUE, GREEN=500, WHITE, BLACK=700 };
```

then `RED` will have the value `100`; `BLUE`, the value `101`; `GREEN`, the value `500`; `WHITE`, the value `501`; and `BLACK`, the value `700`. You can define variables of type `COLOR`, but they can be assigned only one of the enumerated values (in this case, `RED`, `BLUE`, `GREEN`, `WHITE`, or `BLACK`, or else `100`, `101`, `500`, `501`, or `700`). You can assign any color value to your `COLOR` variable. In fact, you can assign any integer value, even if it is not a legal color, although a good compiler will issue a warning if you do. It is important to realize that enumerator variables actually are of type `unsigned int`, and that the enumerated constants equate to integer variables. It is, however, very convenient to be able to name these values when working with colors, days of the week, or similar sets of values. Below is a program that uses an enumerated type.

```
1:  #include <iostream.h>
2:  int main()
3:  {
4:      enum Days { Sunday, Monday, Tuesday, Wednesday, Thursday,
                                    Friday,Saturday };
5:
6:      Days DayOff;
7:      int x;
8:
9:      cout << "What day would you like off (0-6)? ";
10:     cin  >> x;
11:     DayOff = Days(x);
12:
13:     if (DayOff == Sunday || DayOff == Saturday)
14:          cout << "\nYou're already off on weekends!\n";
15:     else
16:          cout << "\nOkay, I'll put in the vacation day.\n";
17:      return 0;
18: }
```

On line 4, the enumerated constant `DAYS` is defined, with seven values counting upward from 0. The user is prompted for a day on line 9. The chosen value, a number between 0 and 6, is compared on line 13 to the enumerated values for Sunday and Saturday, and action is taken accordingly. You cannot type the word "Sunday" when prompted for a day; the program does not know how to translate the characters in `Sunday` into one of the enumerated values.

## Expressions and Statements

At its heart, a program is a set of commands executed in sequence. The power in a program comes from its capability to execute a set of commands, based on whether a particular condition is true or false.

### Statements

In C++ a statement controls the sequence of execution, evaluates an expression, or does nothing (the null statement). A *null statement* is a statement that does nothing. All C++ statements end with a semicolon, even the null statement, which is just the semicolon and nothing else. One of the most common statements is the following assignment statement: `x = a + b;`

Unlike in algebra, this statement does not mean that `x` equals `a+b`. This is read, "Assign the value of the sum of `a` and `b` to `x`," or "Assign to `x`, `a+b`." Even though this statement is doing two things, it is one statement and thus has one semicolon. The assignment operator assigns whatever is on the right side of the equal sign to whatever is on the left side.

#### *Whitespace*

*Whitespace (tabs, spaces, and newlines) is generally ignored in statements.* Whitespace characters *(spaces, tabs, and newlines) cannot be seen. If these characters are printed, you see only the white of the paper. The assignment statement previously discussed could be written as* `x=a+b;` *or as*

```
x       =       a       +               b               ;
```

Although this last variation is perfectly legal, it is also perfectly unwise. Whitespace can be used to make your programs more readable and easier to maintain, or it can be used to create horrific and indecipherable code. In this, as in all things, C++ provides the power; you supply the judgment.

### *Blocks and Compound Statements*

Any place you can put a single statement, you can put a compound statement, also called a block. A block begins with an opening curly brace ({) and ends with a closing curly brace (}). Although every statement in the block must end with a semicolon, the block itself does not end with a semicolon. For example

```
{
      temp = a;
      a = b;
      b = temp;
}
```

This block of code acts as one statement and swaps the values in the variables a and b.

### Expressions

Anything that evaluates to a value is an expression in C++. An expression is said to return a value. Thus, `3+2;` returns the value `5` and so it is an expression. All expressions are statements. There are many pieces of code that qualify as expressions and some might surprise you. Here are three examples:

```
3.2                         // returns the value 3.2
PI                          // float const that returns the value 3.14
SecondsPerMinute            // int const that returns 60
```

Assuming that `PI` is a constant equal to `3.14` and `SecondsPerMinute` is a constant equal to `60`, all three of these statements are expressions. The complicated expression: x = a + b;

not only adds *a* and *b* and assigns the result to *x*, but returns the value of that assignment (the value of *x*) as well. Thus, this statement is also an expression. Because it is an expression, it can be on the right side of an assignment operator: y = x = a + b;

This line is evaluated in the following order: Add *a* to *b*. Assign the result of the expression *a + b* to *x*. Assign the result of the assignment expression *x = a + b to y*. If *a*, *b*, *x*, and *y* are all integers, and if *a* has the value `2` and *b* has the value `5`, both *x* and *y* will be assigned the value `7`. The following program demonstrates the evaluation of complex expressions:

```
1:   #include <iostream.h>
2:   int main()
3:   {
4:      int a=0, b=0, x=0, y=35;
5:      cout << "a: " << a << " b: " << b;
6:      cout << " x: " << x << " y: " << y << endl;
7:      a = 9;
8:      b = 7;
9:      y = x = a+b;
10:     cout << "a: " << a << " b: " << b;
11:     cout << " x: " << x << " y: " << y << endl;
12:       return 0;
13: }
```

On line 4, the four variables are declared and initialized. Their values are printed on lines 5 and 6. On line 7, `a` is assigned the value `9`. One line 8, `b` is assigned the value `7`. On line 9, the values of `a` and `b` are summed and the result is assigned to `x`. This expression (`x = a+b`) evaluates to a value (the sum of `a + b`), and that value is in turn assigned to `y`.

## Operators

An operator is a symbol that causes the compiler to take an action. Operators act on operands, and in C++ all operands are expressions. In C++ there are several different categories of operators such as assignment operators and mathematical operators.

### *Assignment Operator*

The assignment operator (=) causes the operand on the left side of the assignment operator to have its value changed to the value on the right side of the assignment operator. The expression

```
x = a + b;
```

assigns the value that is the result of adding *a* and *b* to the operand *x*. An operand that legally can be on the left side of an assignment operator is called an *lvalue*. That which can be on the right side is called an *rvalue*. Constants are *rvalues*. They cannot be *lvalues*. Thus, you can write

```
x = 35;             // ok
```

but you can't legally write

```
35 = x;             // error, not an lvalue!
```

An *lvalue* is an operand that can be on the left side of an expression. An rvalue is an operand that can be on the right side of an expression. Note that all lvalues are rvalues, but not all rvalues are lvalues. An example of an rvalue that is not an lvalue is a literal. Thus, you can write `x = 5;`, but you cannot write `5 = x;`.

### *Mathematical Operators*

There are five mathematical operators: addition (+), subtraction (-), multiplication (*), division (/), and modulus (%). Addition and subtraction work as you would expect, although subtraction with unsigned integers can lead to surprising results, if the result is a negative number. The following program shows what happens when you subtract a large unsigned number from a small unsigned number.

```
1: // Demonstrating subtraction and
2: // integer overflow
3: #include <iostream.h>
4:
5: int main()
6: {
7:    unsigned int difference;
8:    unsigned int bigNumber = 100;
9:    unsigned int smallNumber = 50;
10:   difference = bigNumber - smallNumber;
11:   cout << "Difference is: " << difference;
12:   difference = smallNumber - bigNumber;
13:   cout << "\nNow difference is: " << difference <<endl;
14:       return 0;
15: }
```

The subtraction operator is invoked on line 10, and the result is printed on line 11, much as we might expect. The subtraction operator is called again on line 12, but this time a large `unsigned` number is subtracted

from a small `unsigned` number. The result would be negative, but because it is evaluated (and printed) as an `unsigned` number, the result is an overflow.

### *Integer Division and Modulus*

Integer division is somewhat different from everyday division. When you divide 21 by 4, the result is a real number (a number with a fraction). Integers don't have fractions, and so the "remainder" is chopped off. The answer is therefore 5. To get the remainder, you take 21 modulus 4 (21 % 4) and the result is 1. The modulus operator tells you the remainder after an integer division. Finding the modulus can be very useful. For example, you might want to print a statement on every 10th action. Any number whose value is 0 when you modulus 10 with that number is an exact multiple of 10. Thus 1 % 10 is 1, 2 % 10 is 2, and so forth, until 10 % 10, whose result is 0. 11 % 10 is back to 1, and this pattern continues until the next multiple of 10, which is 20.

***The lovely students are to be given an assignment at this point !!!***

### Combining the Assignment and Mathematical Operators

It is not uncommon to want to add a value to a variable, and then to assign the result back into the variable. If you have a variable `myAge` and you want to increase the value by two, you can write

```
int myAge = 5;
int temp;
temp = myAge + 2;  // add 5 + 2 and put it in temp
myAge = temp;      // put it back in myAge
```

This method, however, is terribly complicated and wasteful. In C++, you can put the same variable on both sides of the assignment operator, and thus the preceding statement becomes

```
myAge = myAge + 2;
```

which is much better. In algebra this expression would be meaningless, but in C++ it is read as "add two to the value in *myAge* and assign the result to *myAge*." Even simpler to write, but perhaps a bit harder to read is

```
myAge += 2;
```

The self-assigned addition operator (`+=`) adds the *rvalue* to the lvalue and then reassigns the result into the *lvalue*. This operator is pronounced *"plus-equals."* The statement would be read as "`myAge` plus-equals two." If `myAge` had the value `4` to start, it would have `6` after this statement. There are self-assigned subtraction (`-=`), division (`/=`), multiplication (`*=`), and modulus (`%=`) operators as well.

***The lovely student is proded to try out these operators !!!***

### Increment and Decrement

The most common value to add (or subtract) and then reassign into a variable is `1`. In C++, increasing a value by 1 is called incrementing, and decreasing by 1 is called decrementing. There are special operators to perform these actions. The increment operator (`++`) increases the value of the variable by 1, and the decrement operator (`--`) decreases it by 1. Thus, if you have a variable, `C`, and you want to increment it, you would use this statement:

```
C++;                    // Start with C and increment it.
```

This statement is equivalent to the more verbose statement `C = C + 1;` which you learned is also equivalent to the moderately verbose statement `C += 1;`

### *Prefix and Postfix*

Both the increment operator (`++`) and the decrement operator(`--`) come in two varieties: prefix and postfix. The prefix variety is written before the variable name (`++myAge`); the postfix variety is written after (`myAge++`). In a simple statement, it doesn't much matter which one you use, but in a complex statement, when you are incrementing (or decrementing) a variable and then assigning the result to another variable, it matters very much. The prefix operator is evaluated before the assignment, the postfix is evaluated after. The semantics of prefix is this: Increment the value and then fetch it. The semantics of postfix is different: Fetch the value and then increment the original. This can be confusing at first, but if `x` is an integer whose value is `5` and you write `int a = ++x;`

you have told the compiler to increment `x` (making it `6`) and then fetch that value and assign it to *`a`*. Thus, *`a`* is now `6` and *`x`* is now `6`. If, after doing this, you then write `int b = x++;`

you have now told the compiler to fetch the value in *`x (6)`* and assign it to *`b`*, and then go back and increment *`x`*. Thus, *`b`* is now `6`, but *`x`* is now `7`. The program below shows the use and implications of both types.

```
1:  // Demonstrating use of
2:  // prefix and postfix increment and
3:  // decrement operators
4:  #include <iostream.h>
5:  int main()
6:  {
7:     int myAge = 39;      // initialize two integers
8:     int yourAge = 39;
9:     cout << "I am: " << myAge << " years old.\n";
10:    cout << "You are: " << yourAge << " years old\n";
11:    myAge++;         // postfix increment
12:    ++yourAge;       // prefix increment
13:    cout << "One year passes...\n";
14:    cout << "I am: " << myAge << " years old.\n";
15:    cout << "You are: " << yourAge << " years old\n";
16:    cout << "Another year passes\n";
17:    cout << "I am: " << myAge++ << " years old.\n";
18:    cout << "You are: " << ++yourAge << " years old\n";
19:    cout << "Let's print it again.\n";
20:    cout << "I am: " << myAge << " years old.\n";
21:    cout << "You are: " << yourAge << " years old\n";
22:      return 0;
23: }
```

On lines 7 and 8, two integer variables are declared, and each is initialized with the value `39`. Their values are printed on lines 9 and 10. On line 11, *`myAge`* is incremented using the postfix increment operator, and on line 12, *`yourAge`* is incremented using the prefix increment operator. The results are printed on lines 14 and 15, and they are identical (both `40`). On line 17, `myAge` is incremented as part of the printing statement, using the postfix increment operator. Because it is postfix, the increment happens after the print,

and so the value `40` is printed again. In contrast, on line 18, *`yourAge`* is incremented using the prefix increment operator. Thus, it is incremented before being printed, and the value displays as `41`. Finally, on lines 20 and 21, the values are printed again. Because the increment statement has completed, the value in *`myAge`* is now `41`, as is the value in *`yourAge`*.

### *Precedence*

In the complex statement `x = 5 + 3 * 8;`

which is performed first, the addition or the multiplication? If the addition is performed first, the answer is 8 * 8, or 64. If the multiplication is performed first, the answer is 5 + 24, or 29. Every operator has a precedence value. Multiplication has higher precedence than addition, and thus the value of the expression is 29. When two mathematical operators have the same precedence, they are performed in left-to-right order. Thus `x = 5 + 3 + 8 * 9 + 6 * 4;`

is evaluated multiplication first, left to right. Thus, 8*9 = 72, and 6*4 = 24. Now the expression is essentially

```
x = 5 + 3 + 72 + 24;
```

Now the addition, left to right, is 5 + 3 = 8; 8 + 72 = 80; 80 + 24 = 104. Be careful with this. Some operators, such as assignment, are evaluated in right-to-left order! In any case, what if the precedence order doesn't meet your needs? Consider the expression

```
TotalSeconds = NumMinutesToThink + NumMinutesToType * 60
```

In this expression, you do not want to multiply the *`NumMinutesToType`* variable by 60 and then add it to *`NumMinutesToThink`*. You want to add the two variables to get the total number of minutes, and then you want to multiply that number by 60 to get the total seconds. In this case, you use parentheses to change the precedence order. Items in parentheses are evaluated at a higher precedence than any of the mathematical operators. Thus

```
TotalSeconds = (NumMinutesToThink + NumMinutesToType) * 60
```

will accomplish what you want.

### Nesting Parentheses

For complex expressions, you might need to nest parentheses one within another. For example, you might need to compute the total seconds and then compute the total number of people who are involved before multiplying seconds times people:

```
TotalPersonSeconds = ( ( (NumMinutesToThink + NumMinutesToType) * 60)
*  (PeopleInTheOffice + PeopleOnVacation) )
```

This expression is read from the inside out. First, *`NumMinutesToThink`* is added to *`NumMinutesToType`*, because these are in the innermost parentheses. Then this sum is multiplied by 60. Next, *`PeopleInTheOffice`* is added to *`PeopleOnVacation`*. Finally, the total number of people found is multiplied by the total number of seconds.

## The Nature of Truth

In C++, zero is considered false, and all other values are considered true, although true is usually represented by 1. Thus, if an expression is false, it is equal to zero, and if an expression is equal to zero, it is false. If a statement is true, all you know is that it is nonzero, and any nonzero statement is true.

### *Relational Operators*

The relational operators are used to determine whether two numbers are equal, or if one is greater or less than the other. Every relational statement evaluates to either `1` (`TRUE`) or `0` (`FALSE`). If the integer variable *`myAge`* has the value `39`, and the integer variable *`yourAge`* has the value `40`, you can determine whether they are equal by using the relational "equals" operator:

```
myAge == yourAge;  // is the value in myAge the same as in yourAge?
```

This expression evaluates to `0`, or `false`, because the variables are not equal. The expression

```
myAge > yourAge;  // is myAge greater than yourAge?
```

evaluates to `0` or `false`. Please note that the assignment operator is the = sign while the equals operator is the ==. There are six relational operators: equals (==), less than (<), greater than (>), less than or equal to (<=), greater than or equal to (>=), and not equals (!=). The table below shows each relational operator and how it is used.

| ***Name*** | ***Operator*** | ***Sample*** | ***Evaluates*** |
|---|---|---|---|
| Equals | == | `100 == 50;` | `false` |
| | | `50 == 50;` | `true` |
| Not Equals | != | `100 != 50;` | `true` |
| | | `50 != 50;` | `false` |
| Greater Than | > | `100 > 50;` | `true` |
| | | `50 > 50;` | `false` |
| Greater Than or Equals | >= | `100 >= 50;` | `true` |
| | | `50 >= 50;` | `true` |
| Less Than | < | `100 < 50;` | `false` |
| | | `50 < 50;` | `false` |
| Less Than or Equals | <= | `100 <= 50;` | `false` |
| | | `50 <= 50;` | `true` |

## The if Statement

Normally, your program flows along line by line in the order in which it appears in your source code. The `if` statement enables you to test for a condition (such as whether two variables are equal) and branch to different parts of your code, depending on the result. The simplest form of an `if` statement is this:

```
if (expression)
     statement;
```

The expression in the parentheses can be any expression, but it usually contains one of the relational expressions. If the expression has the value `0`, it is considered false, and the statement is skipped. If it has any nonzero value, it is considered true, and the statement is executed. Consider the following example:

```
if (bigNumber > smallNumber)
     bigNumber = smallNumber;
```

This code compares *`bigNumber`* and *`smallNumber`*. If *`bigNumber`* is larger, the second line sets its value to the value of *`smallNumber`*.

A block of statements surrounded by braces is exactly equivalent to a single statement, here's a simple example of this usage:

```
if (bigNumber > smallNumber)
{
     bigNumber = smallNumber;
     cout << "bigNumber: " << bigNumber << "\n";
     cout << "smallNumber: " << smallNumber << "\n";
}
```

This time, if *`bigNumber`* is larger than *`smallNumber`*, not only is it set to the value of *`smallNumber`*, but an informational message is printed. The following program shows a more detailed example of branching based on relational operators.

```
1:  // Program to demonstrate the if statement
2:  // used with relational operators
3:  #include <iostream.h>
4:  int main()
5:  {
6:      int TeamAScore, TeamBScore;
7:      cout << "Enter the score for Team A: ";
8:      cin >> TeamAScore;
9:
10:     cout << "\nEnter the score for Team B: ";
11:     cin >> TeamBScore;
12:
13:     cout << "\n";
14:
15:     if (TeamAScore > TeamBScore)
16:         cout << "Bravo Team A!\n";
17:
18:     if (TeamAScore < TeamBScore)
19:     {
20:         cout << "Bravo Team B!\n";
21:         cout << "Happy days in Shangombo!\n";
22:     }
23:
24:     if (TeamAScore == TeamBScore)
25:     {
26:         cout << "A tie? Naah, can't be.\n";
27:         cout << "Give me the real score for the Shangis (Team B): ";
28:         cin >> TeamBScore;
29:
```

```
30:         if (TeamAScore > TeamBScore)
31:             cout << "Knew it! Bravo Team A!";
32:
33:         if (TeamBScore > TeamAScore)
34:             cout << "Knew it! Bravo Team B – The Shangis!";
35:
36:         if (TeamBScore == TeamAScore)
37:             cout << "Wow, it really was a tie!";
38:     }
39:
40:     cout << "\nThanks for telling me about the scores.\n";
41:      return 0;
42: }
```

This program asks for user input of scores for two baseball teams, which are stored in integer variables. The variables are compared in the `if` statement on lines 15, 18, and 24. If one score is higher than the other, an informational message is printed. If the scores are equal, the block of code that begins on line 24 and ends on line 38 is entered. The second score is requested again, and then the scores are compared again. Note that if the initial Team B score was higher than Team A score, the `if` statement on line 15 would evaluate as `FALSE`, and line 16 would not be invoked. The test on line 18 would evaluate as `true`, and the statements on lines 20 and 21 would be invoked. Then the `if` statement on line 24 would be tested, and this would be false (if line 18 was true). Thus, the program would skip the entire block, falling through to line 39. In this example, getting a true result in one `if` statement does not stop other `if` statements from being tested.

### *The Else Statement*

Often your program will want to take one branch if your condition is true and another if it is false. In In the previous program we wanted to print one message (Bravo Team A!) if the first test (TeamAScore > TeamBScore) evaluated `TRUE`, and another message (Bravo Team B!) if it evaluated `FALSE`. The method shown so far, testing first one condition and then the other, works fine but is a bit cumbersome. The keyword *`else`* can make for far more readable code as given below.

```
if (expression)
     statement;
else
     statement;
```

### The if Statement

The syntax for the `if` statement can take many forms as is shown below: Form 1

```
if (expression)
    statement;
next statement;
```

If the expression is evaluated as `TRUE`, the statement is executed and the program continues with the next statement. If the expression is not true, the statement is ignored and the program skips to the next statement. Remember that the statement can be a single statement ending with a semicolon or a block enclosed in braces. Form 2

```
if (expression)
    statement1;
else
    statement2;
next statement;
```

If the expression evaluates `TRUE`, `statement1` is executed; otherwise, `statement2` is executed. Afterwards, the program continues with the next statement. This is shown in the next example.

```
Example
if (SomeValue < 10)
  cout << "SomeValue is less than 10");
else
  cout << "SomeValue is not less than 10!");
cout << "Done." << endl;
```

### *Advanced if Statements*

It is worth noting that any statement can be used in an `if` or `else` clause, even another `if` or `else` statement. Thus, you might see complex `if` statements in the following form:

```
if (expression1)
{
    if (expression2)
        statement1;
    else
    {
        if (expression3)
            statement2;
        else
            statement3;
    }
}
else
    statement4;
```

This cumbersome `if` statement says, "If expression1 is true and expression2 is true, execute statement1. If expression1 is true but expression2 is not true, then if expression3 is true execute statement2. If expression1 is true but expression2 and expression3 are false, execute statement3. Finally, if expression1 is not true, execute statement4." As can be seen, complex `if` statements can be confusing!

### Using Braces in Nested if Statements

Although it may be legal to leave out the braces on `if` statements that are only a single statement, and it is legal to nest `if` statements, such as

```
if (x > y)                 // if x is larger than y
    if (x < z)             // and if x is smaller than z
        x = y;            // then set x to the value in z
```

when writing large nested statements, this can cause enormous confusion. Remember, whitespace and indentation are a convenience for the programmer; they make no difference to the compiler. It is easy to confuse the logic and inadvertently assign an `else` statement to the wrong `if` statement.

## Logical Operators

Often you want to ask more than one relational question at a time. *"Is it true that x is greater than y, and also true that y is greater than z?"* A program might need to determine that both of these conditions are true, or that some other condition is true, in order to take an action. Imagine a sophisticated alarm system that has this logic: *"If the door alarm sounds AND it is after 18:00 hour AND it is NOT a holiday, OR if it is a weekend, then call the police."* C++'s three logical operators are used to make this kind of evaluation. These operators are listed in the table below:

| ***Operator*** | ***Symbol*** | ***Example*** |
|---|---|---|
| AND | && | expression1 && expression2 |
| OR | \|\| | expression1 \|\| expression2 |
| NOT | ! | !expression |

### *Logical AND*

A logical `AND` statement evaluates two expressions, and if both expressions are true, the logical `AND` statement is true as well. If it is true that you are hungry, AND it is true that you have money, THEN it is true that you can buy lunch. Thus,

```
if ( (x == 5) && (y == 5) )
```

would evaluate `TRUE` if both `x` and `y` are equal to 5, and it would evaluate `FALSE` if either one is not equal to 5. Note that both sides must be true for the entire expression to be true. Note that the logical `AND` is two `&&` symbols. A single `&` symbol is a different operator altogether.

### *Logical OR*

A logical `OR` statement evaluates two expressions. If either one is true, the expression is true. If you have money in cash OR you have a credit card, you can pay the bill. You don't need both money in cash and a credit card; you need only one, although having both would be fine as well. Thus,

```
if ( (x == 5) || (y == 5) )
```

evaluates `TRUE` if either `x` or `y` is equal to 5, or if both are. Note that the logical `OR` is two `||` symbols. A single `|` symbol is a different operator altogether.

### *Logical NOT*

A logical `NOT` statement evaluates `true` if the expression being tested is false. Again, if the expression being tested is false, the value of the test is `TRUE`! Thus `if ( !(x == 5) )`

is true only if `x` is not equal to 5. This is exactly the same as writing `if (x != 5)`

## Relational Precedence

Relational operators and logical operators, being C++ expressions, each return a value: `1` (`TRUE`) or `0` (`FALSE`). Like all expressions, they have a precedence order that determines which relations are evaluated first. This fact is important when determining the value of the statement

```
if ( x > 5 &&  y > 5  || z > 5)
```

It might be that the programmer wanted this expression to evaluate `TRUE` if both `x` and `y` are greater than 5 or if `z` is greater than 5. On the other hand, the programmer might have wanted this expression to evaluate `TRUE` only if `x` is greater than 5 and if it is also true that either `y` is greater than 5 or `z` is greater than 5. If `x` is 3, and `y` and `z` are both 10, the first interpretation will be true (`z` is greater than 5, so ignore `x` and `y`), but the second will be false (it isn't true that both `x` and `y` are greater than 5 nor is it true that `z` is greater than 5). Although precedence will determine which relation is evaluated first, parentheses can both change the order and make the statement clearer:

```
if (  (x > 5)  && (y > 5 ||  z > 5) )
```

Using the values from the earlier example, this statement is false. Because it is not true that `x` is greater than 5, the left side of the `AND` statement fails, and thus the entire statement is false. Remember that an `AND` statement requires that both sides to be true. It is often a good idea to use extra parentheses to clarify what you want to group. Remember, the goal is to write programs that work and that are easy to read and understand.

## More About Truth and Falsehood

In C++, zero is false, and any other value is true. Because an expression always has a value, many C++ programmers take advantage of this feature in their `if` statements. A statement such as

```
if (x)            // if x is true (nonzero)
    x = 0;
```

can be read as "If `x` has a nonzero value, set it to 0." This is a bit unclear; it would be clearer if written

```
if (x != 0)       // if x is nonzero
    x = 0;
```

Both statements are legal, but the latter is clearer. It is good programming practice to reserve the former method for true tests of logic, rather than for testing for nonzero values. The following two statements also are equivalent:

```
if (!x)            // if x is false (zero)
if (x == 0)        // if x is zero
```

The second statement, however, is somewhat easier to understand and is more explicit. It is also common practice to define your own enumerated Boolean (logical) type with `enum Bool {FALSE, TRUE};`. This serves to set `FALSE` to `0` and `TRUE` to `1`.

## Conditional (Ternary) Operator

The conditional operator (`?:`) is C++'s only ternary operator; that is, it is the only operator to take three terms.

The conditional operator takes three expressions and returns a value:

```
(expression1) ? (expression2) : (expression3)
```

This line is read as "If expression1 is true, return the value of expression2; otherwise, return the value of expression3." Typically, this value would be assigned to a variable. The following program gives an example of the use of conditional operators.

```
1:  // Demonstrating the conditional operators
2:  //
3:  #include <iostream.h>
4:  int main()
5:  {
6:     int x, y, z;
7:     cout << "Enter two numbers.\n";
8:     cout << "First: ";
9:     cin >> x;
10:    cout << "\nSecond: ";
11:    cin >> y;
12:    cout << "\n";
13:
14:    if (x > y)
15:       z = x;
16:    else
17:       z = y;
18:
19:    cout << "z: " << z;
20:    cout << "\n";
21:
22:    z =  (x > y) ? x : y;
23:
24:    cout << "z: " << z;
25:    cout << "\n";
26:       return 0;
27: }
```

Three integer variables are created: x, y, and z. The first two are given values by the user. The if statement on line 14 tests to see which is larger and assigns the larger value to z. This value is printed on line 19. The conditional operator on line 22 makes the same test and assigns z the larger value. It is read like this: "If x is greater than y, return the value of x; otherwise, return the value of y." The value returned is assigned to z. That value is printed on line 24. As you can see, the conditional statement is a shorter equivalent to the if...else statement.