

A statement is just an instruction, essentially it's an imperative sentence.

Here are some examples of C++ statements:

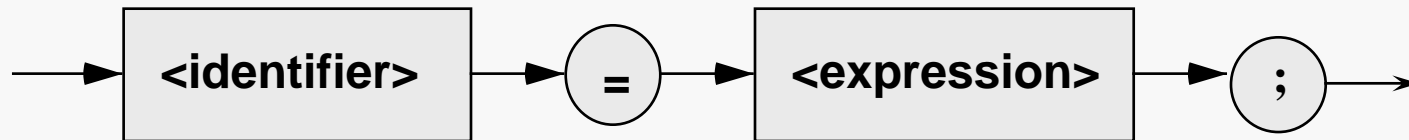
```
const float PI = 3.141596F;  
double GPA;  
GPA = totalCreditPoints / totalCreditHours;  
cout << "My name is "  
      << userName << endl;
```

C++ statements may be on a single line, or they may span several lines.

Just as every English sentence must end with a punctuation mark, every C++ statement must end with a symbol to mark its end.

In C++, every statement must end with a semicolon ' ; '.

Syntax:



In C++, the symbol '=' is the assignment operator; it does NOT represent equality!

Example: `totalScore = totalScore + newPoints;`

An assignment statement is an executable statement. It gives a variable the value of an expression.

Semantics: the value of the expression on the right side is stored in the memory location referenced by the variable on the left side, `totalScore` in this case.

Remember that in C++ the symbol '=' does not represent equality, as it does in mathematics. The statement:

$$X = X + 1;$$

calculate the value $X + 1$ (using the current value of X) and store the result in X .

It is good practice to make sure that the type of the expression is the same as the type of the "receiving" variable. However, that is not (perhaps unfortunately) required in C++:

Consider the previous declarations (3.12). In these assignments, the type of the source and the type of the target match:

```
Weight          =      123;      // Fine, int <--- int
GPA              =      3.5;      // Fine, double <--- double
MiddleInitial    =      'L';      // Fine, char <--- char
Major           =      "MATH";    // Fine, string <--- string
```

As long as the type of the source and target are the same the effect of the assignment is simple.

Of course, some assignments are illogical even though the types do match:

```
Height = Weight; // Doesn't make logical sense. . .
```

The compiler will not object to this because the logical meaning of the variables is known only to the human programmer.

It is good practice to include descriptive comments in program source code.

Comments may explain the purpose of a declared identifier, or of a statement or group of statements that perform some calculation, or input or output.

There are two different syntaxes for comments in C++:

```
int quizScore;           // score on a quiz
int numQuizzes = 0;      // number of quizzes given
int totalPoints;        // sum of all quiz scores
double quizAverage;     // average of all quiz scores

/* Read in quiz scores until the user enters
   one that's negative. */
cin >> quizScore;
while (quizScore >= 0) {
    totalPoints = totalPoints + quizScore;
    numQuizzes = numQuizzes + 1;
    cin >> quizScore;
}
// Calculate average quiz score:
quizAverage = double(totalPoints) / numQuizzes;
```

C++ uses the following symbols to represent the basic arithmetic operations:

Symbol	Meaning	Example	Value
+	addition	43 + 8	51
-	subtraction	43.0 - 8.0	35.0
*	multiplication	43 * 8	344
/	division	43.0 / 8.0	5.375
%	remainder	43 % 8	3
-	unary minus	-43	-43

The operands can be of any type for which the operation makes sense.

Parentheses may be used to group terms in more complex expressions:

$$32 - (3 + 2) * 5 \quad \text{---->} \quad 32 - 5 * 5 \quad \text{---->} \quad 32 - 25 \quad \text{---->} \quad 7$$

There is no operator in C++ for exponentiation (x^y).

Precedence rules determine the order in which operations are performed:

0. Expressions grouped in parentheses are evaluated first.
1. unary -
2. *, /, and %
3. + and -

Operators within the same level are evaluated in left-to-right order (w/o parentheses).

These are the same rules used in mathematical expressions, so they should feel natural.

When in doubt, add parentheses for clarity!

Examples:

$24 / 7 + 5$	---->	$3 + 5$	---->	8		
$24 / (7 + 5)$	---->	$24 / 12$	---->	2		
$1 + 2 - 3 - 4$	---->	$3 - 3 - 4$	---->	$0 - 4$	---->	-4
$4 * 3 - 2$	---->	$12 - 2$	---->	10		

When the type of the source is NOT the same as the type of the target variable, errors may result.

When a decimal value is assigned to an integer variable, the decimal value is automatically truncated to an integer value when it is stored:

```
const int NUMTESTS = 2;
double Test1 = 93.0,
       Test2 = 86.0;
int     testAverage;
testAverage = (Test1 + Test2)/NUMTESTS; // testAverage <--- 89
                                         //      not 89.5
```

When an integer value is assigned to a decimal variable, the integer value is automatically "widened" to a decimal value when it is stored:

```
double Sum;
int X = 17, Y = 25, Z = 42;
Sum = X + Y + Z;           // Sum <--- 84.0, not 84
```

Even though 84 and 84.0 are the same number, from a mathematical perspective, they are not stored the same way on your computer.

When the type of the source is NOT the same as the type of the target variable, and that is deliberate, the programmer may explicitly specify the conversion:

```
const double PI = 3.141596224;
double Radius = 1.432;
int    Circumference;
Circumference = int(2.0 * PI * Radius);
```

The expression `int(someExpression)` indicates that the value of `someExpression` is to be converted to an integer. This is known as an explicit typecast.

Using an explicit typecast doesn't eliminate any logical errors, but it does indicate that the programmer is aware that the conversion is taking place. In the example above, writing the explicit conversion just might warn the programmer that a logical error is probably being committed, since the circumference of a circle is generally a decimal value.

Explicit typecasts are accomplished by using conversion operators. A conversion operator is an expression of the form `type ()` where `type` can be any built-in data type.

Some conversions are not supported. For example: `string(17.2)`

Decimal values and integers can be combined using all arithmetic operators.

Integer values are converted to decimal and resulting expression type is float or double, as appropriate:

```
float X = 9.0 / 12;           // value 0.75 is stored
double Y = 5 / 2.0;          // value 2.5 is stored
int    A = 5,
       B = 2;
double Z = A / B;            // value 2.0 is stored (Why?)
```

The last example might benefit from the use of a conversion operator:

```
double Z = double(A) / double(B); // value 2.5 is stored
```

The use of a variable as a counter is so common that C++ provides a shorthand notation for adding or subtracting 1:

```
int x = 0,
    y = 7;
x++;           // same effect as x = x + 1;
y--;           // same effect as y = y - 1;
```

The notation `x++` is referred to as postfix increment since the operator `++` comes after the variable being incremented. The operator may also be used as a prefix: `++x`.

These two statements have exactly the same effect:

```
int x = 0;
x++;
++x;
```

But it's more complex when the increment/decrement operators are used in a larger expression::

```
int x = 0, y = 7, z;
z = y * x++;           // z <--- 7 * 0
z = y * ++x;           // z <--- 7 * 1
```

Semantically the prefix and postfix versions of the increment operator are different. To understand, remember that `++x` is itself an expression.

Semantic Rules:

- prefix incrementation (`++x`) means that the variable is incremented before the value of the expression is determined.
- postfix incrementation (`x++`) means that the variable is incremented after the value of the expression is determined.

So:

```
int x = 0, y = 7, z;

z = y * x++;           // 7 * 0 is evaluated, and THEN
                       // x <--- 1, and THEN z <--- 0

z = y * ++x;          // x <--- 1, then THEN 7 * 1 is
                       // evaluated, and THEN z <--- 7
```

It's better design to avoid using these operators within a larger expression.

It is often necessary to group statements together (like a paragraph in a term paper).

In C++ this is accomplished by surrounding a collection of statements with "curly braces":

```
int main() {                // begin compound stmt
    int x = 42;
    cout << "x = " << x;
    return 0;
}                            // end compound stmt
```

Curly braces **MUST** come in matched pairs. The rule is that a closing brace '}' matches the closest preceding unmatched opening brace '{'.

Compound statements are terminated by the closing brace, **NOT** by a semicolon.

Syntax:



Two strings may be concatenated; that is, one may be appended to another:

```
string Greet1    = "Hello";  
string Greet2    = "world";  
string Greetings = Greet1 + ", " + Greet2 + '!';
```

Here, the concatenation operator (+) is used to combine two string variables, a literal string, and a literal character; the result is assigned to Greetings.

It is not legal to have a line break occur within a literal string:

```
string BadString = "It is as a tale told by an idiot, // not  
                    full of sound and fury,           // legal  
                    signifying nothing.";
```

However long initializations may be broken across lines like this:

```
string LongString = "It is as a tale told by an idiot, "  
                    "full of sound and fury, "  
                    "signifying nothing.";
```

Provide some additional operations on specific data types. Functions are called (or invoked) via expressions of the form:

function_name (parameter_list)

where each parameter can be an arithmetic expression.

```
// Determine the difference between two integers, a and b
int Diff = abs(a - b);           // <cmath>

// Get the length of a string
string Proverb = "There is always time to do it over";

int Len = Proverb.length();     // <string>

// Get the square root of a number
double rootX = sqrt(X);        // <cmath>
```

The use of standard library functions requires the inclusion of specific standard header files, such as `cmath`, and a `using namespace` directive.

A function invocation temporarily interrupts the default sequential statement-to-statement *flow of control*:

```
#include <iostream>
#include <cmath>
using namespace std;
int main() {
    int First, Second, Diff;
    cout << "Please enter two integers: ";
    cin >> First >> Second;

    Diff = abs( First - Second );

    cout << "The difference is "
         << Diff << endl;
    return 0;
}
```

Implementation of abs() is in the C++ Standard Library, not written as part of this program.

```
// implementation of
//      function abs

int abs(int Value) {
    int absValue;
    if (Value < 0)
        absValue = -Value;
    else
        absValue = Value;
    return absValue;
}
```

Normal flow of control resumes after the called function completes its execution (returns).