• Assignment statement, its types/ with explanation examples • Arithmetic expression (equation) • counters, counter types • deferent images for equations belong to C++ language	السادس

Assignment statement

Operators

Once introduced to variables and constants, we can begin to operate with them by using *operators*. What follows is a complete list of operators. At this point, it is likely not necessary to know all of them, but they are all listed here to also serve as reference.

Assignment operator (=)

The assignment operator assigns a value to a variable.

```
x = 5;
```

This statement assigns the integer value 5 to the variable x. The assignment operation always takes place from right to left, and never the other way around:

```
x = y;
```

This statement assigns to variable x the value contained in variable y. The value

of x at the moment this statement is executed is lost and replaced by the value of y.

Consider also that we are only assigning the value of y to x at the moment of the assignment operation. Therefore, if y changes at a later moment, it will not affect the new value taken by x.

For example, let's have a look at the following code - I have included the evolution of the content stored in the variables as comments:

```
// assignment operator
#include <iostream>
using namespace std;

int main ()
{
  int a, b;       // a:?,  b:?
  a = 10;         // a:10, b:?
  b = 4;          // a:10, b:4
  a = b;          // a:4,  b:4
  b = 7;          // a:4,  b:7

  cout << "a:";
  cout << a;
  cout << " b:";
  cout << b;
}
```

```
a:4 b:7
```

This program prints on screen the final values of a and b (4 and 7, respectively). Notice how a was not affected by the final modification of b, even though we declared a = b earlier.

Assignment operations are expressions that can be evaluated. That means that the assignment

itself has a value, and -for fundamental types- this value is the one assigned in the operation. For example:

```
y = 2 + (x = 5);
```

In this expression, y is assigned the result of adding 2 and the value of another assignment expression (which has itself a value of 5). It is roughly equivalent to:

```
1 x = 5;
2 y = 2 + x;
```

With the final result of assigning 7 to y.

The following expression is also valid in C++:

```
x = y = z = 5;
```

It assigns 5 to the all three variables: x, y and z; always from right-to-left.

Arithmetic operators (+, -, *, /, %)

The five arithmetical operations supported by C++ are:

operator	description
+	addition
-	subtraction
*	multiplication
/	division

%	modulo

Operations of addition, subtraction, multiplication and division correspond literally to their respective

mathematical operators. The last one, *modulo operator*, represented by a percentage sign (%), gives the remainder of a division of two values. For example:

```
x = 11 % 3;
```

results in variable x containing the value 2, since dividing 11 by 3 results in 3, with a remainder of 2.

Compound assignment (+=, -=, *=, /=, %=, >>=, <<=, &=, ^=, |=)

Compound assignment operators modify the current value of a variable by performing an operation on it. They are equivalent to assigning the result of an operation to the first operand:

expression	equivalent to...
y += x;	y = y + x;
x -= 5;	x = x - 5;
x /= y;	x = x / y;
price *= units + 1;	price = price * (units+1);

and the same for all other compound assignment operators. For example:

```
// compound assignment operators
#include <iostream>
using namespace std;

int main ()
{
 int a, b=3;
```

```
5
```

```
 a = b;
 a+=2;          // equivalent to a=a+2
 cout << a;
}
```

Counters

Increment and decrement (++, --)

Some expression can be shortened even more: the increase operator (++) and the decrease operator (--) increase or reduce by one the value stored in a variable. They are equivalent to +=1 and to -=1, respectively. Thus:

```
++x;
x+=1;
x=x+1;
```

are all equivalent in its functionality; the three of them increase by one the value of x.

In the early C compilers, the three previous expressions may have produced different executable code depending on which one was used. Nowadays, this type of code optimization is generally performed automatically by the compiler, thus the three expressions should produce exactly the same executable code.

A peculiarity of this operator is that it can be used both as a prefix and as a suffix. That means that it can be written either before the variable name (++x) or after it (x++). Although in simple expressions like x++ or ++x, both have exactly the same meaning; in other expressions in which the result of the increment or decrement operation is evaluated, they may have an important difference in their meaning: In the case that the increase operator is used as a prefix (++x) of the value, the expression evaluates to the final value of x, once it is already increased. On the other hand, in case that it is used as

a suffix (x++), the value is also increased, but the expression evaluates to the value that x had before being increased. Notice the difference:

Example 1	**Example 2**

x = 3;
y = ++x;
// x contains 4, y contains 4
	x = 3;
	y = x++;
	// x contains 4, y contains 3

In *Example 1*, the value assigned to y is the value of x after being increased. While in *Example 2*, it is the value x had before being increased.