Introdução à Programação 2020/2021

António J. R. Neves
Daniel Corujo

Departamento de Electrónica, Telecomunicações e Informática
Universidade de Aveiro

an@ua.pt / dcorujo@ua.pt

http://elearning.ua.pt/

- Assignment operator
- Increment and decrement operators
- **`while`** statement
- **`do ... while`** statement
- **`for`** statement

- In a simple assignment (performed using the operator =), the assignment operation stores the value of the expression on the right hand-side of the = operator in the variable whose name appears on the left hand side of the = operator.
- The assignment itself is an **expression**, that can be used inside a more complex expression (use parentheses if necessary to make your intentions crystal clear):

```
int x, y, z;
x = 3;              // makes the current value of x
                    //     equal to 3
z = y = x;          // same as z = (y = x);
                    // makes z and y equal to x
z = x + (y = 3); // makes y equal to 3
                    // and z equal to x + 3
```

- There are also compound assignments, which combine an arithmetic operation or a bitwise logic operation in the same step with the assignment (example: `+=`, `-=` `*=`, `/=`, `%=`, `...`).
- In evaluating a compound assignment expression, the program combines the two operands with the specified operation and assigns the result to the left operand.
- The value of the entire assignment expression is the same as the value assigned to the left operand, and the assignment expression has the type of the left operand.

```
int x = 1, y = 2, z = 3;
x += 5;          // equivalent to x = x + 5;
y *= 5 + x;      // equivalent to y = y * (5 + x);
z /= x + y;      // equivalent to z = z / (x + y)
```

- The tokens `++` and `--` can be either postfix or prefix operators.
  - `++` increments, i.e., adds one (`++x`, `x++`)
  - `--` decrements, i.e., subtracts one (`--x`, `x--`)
- The value of `x++` is the value that `x` had before it was incremented; the value of `x--` is the value that `x` had before it was decremented.
- The value of `++x` is the value that `x` has after it has been incremented; The value of `--x` is the value that `x` has after it has been decremented.
- `++x++`, `++x--`, `--x++`, and `--x--` are **illegal**.
- The result of `x++ + x++` (and of other expressions of this kind that increment or decrement the same variable more than once), is i**mplementation dependent**.

- Consider the following code fragment:

```
int x = 2, y;
// x = 2
// y = ??? (undefined)
y = 2 * x++ + 3;
// x = 3
// y = 7


// same as
//    y = 2 * x + 3;
//    x = x + 1;
```

```
int x = 2, y;
// x = 2
// y = ??? (undefined)
y = 3 + ++x * 2;
// x = 3
// y = 9


// same as
//    x = x + 1;
//    y = 3 * x + 2;
```

- In a computer program it is often necessary to perform a certain action a certain number of times or until a certain condition is met.
- The constructs that enable computers to perform certain repetitive tasks are called **loops**.
- We can start a loop using one of three iteration statements: **`while`**, **`do ... while`**, and **`for`**.
- In each of these statements, the number of iterations through the loop body is controlled by a condition, the so-called controlling expression.

```
while(condition){
    statements;
}
```

```
do{
    statements;
}
while(condition);
```

- A **while** statement executes a statement repeatedly as long as the controlling expression is true.
- The while statement is a top-driven loop:
  - first, the controlling expression is evaluated.
  - If it evaluates to true, the loop body is executed, and then the controlling expression is evaluated again.
  - If it evaluates to false, the program execution continues with the statement that follows the loop body.

- The **`do ... while`** statement is a bottom-driven loop.
- The loop body statement is executed once before the controlling expression is evaluated for the first time.
- If the controlling expression evaluates to true, then another iteration follows. If it evaluates to false, the loop is finished.
- Unlike the while and for statements, **`do ... while`** ensures that at least one iteration of the loop body is performed.
- **Nested Loops**: A loop body can be any simple or block statement, and may include other loop statements.

Example of a program fragment that can be used for reading a positive number (**do** … **while** loop):

```
int x, cont = 0;
do{
      cout << "Enter a positive value: ";
      cin >> x;
      cont++;
}while(x <= 0);

cout << "value " << x << " read in " << cont << "attempts";
```

Example of a program fragment that can be used for reading a positive number (**while** loop):

```
int x = -1;
int cont = 0;
// another option is to read a first value outside the loop
while(x <= 0){
      cout << "Enter a positive value: ";
      cin >> x;
      cont++;
}

cout << "value " << x << " read in " << cont << "attempts";
```

- Like the **`while`** statement, the **`for`** statement is a top-driven loop, but with more loop logic contained within the statement itself:

```
for(expr1 ; expr2 ; expr3)
    statement(s)
```

- The three actions that need to be executed in a typical loop are specified together at the top of the loop body:
  - **`expr1 (initialization)`**

    Evaluated only once, before the first evaluation of the controlling expression, to perform any necessary initialization; may be empty (do nothing);

  - **`expr2 (controlling expression)`**

    Tested before each iteration. Loop execution ends when this expression evaluates to false; may be empty (always true);

  - **`expr3 (adjustment)`**

    An adjustment, such as incrementing a counter, performed after each loop iteration and before **`expr2`** is tested again; may be empty (do nothing).

The code

```
for(A ; B ; C)
  D;
```

where **A**, **B**, and **C** are expressions, and **D** is a block of code is equivalent to the code

```
A;
while(B){
  D;
  C;
}
```

Usually **A** and **C** contain initialization or updates of one or more variables (use a comma to separate sub-expressions), **B** is the loop controlling expression, and **D** is a block of code.

Example

```
int i, n;
do{
    cout << "Multiplication table of: ";
    cin >> n;
}while(n < 1 || n > 10);

for(i = 1 ; i <= 10 ; i++)
        cout << n << " x " << i << " = " << n* i <<
  endl;
```

- The **`break`** and **`continue`** loop control statements change the way the rest of the loop is executed:
  - The **`break`** statement terminates the loop and transfers execution to the statement immediately following the loop. It can occur only in the body of a loop or of a switch statement.
  - The **`continue`** statement can be used only within the body of a loop. It skips the rest of the loop body, jumping immediately to the controlling expression in **`while`** and **`do ... while`** loops and jumping to the updating expression (**`expr3`** in a previous slide), in a **`for`** loop.

Example

```
int x, cont = 0;
do{
    cout << "Introduza um valor inteiro positivo: ";
    cin >> x;
    cont++;
    if(cont >= 10)  // after 10 attempts, terminate the loop
        break;
}while(x <= 0);

if(x > 0)
  cout << "Value " << x << " read in " << cont << " attempts";
else
  cout << "Unable to read a value in ten attempts" << endl;
```

Another example

```
int i, n, sum = 0;
do{
    cout << "Value of N [1 ... 100]: ";
    cin >> n;
}while(n < 1 || n > 100);

for(i = 1 ; i <= n ; i++){
    // if an even number, advance to the next integer
    if(i % 2 == 0)
        continue;

    // accumulate (add)
    sum += i;
}

cout << "The sum of the odd numbers is " << sum << endl;
```

Another example (more efficient)

```
int i, n, sum;
do{
     cout << "Value of N [1 ... 100]: ";
     cin >> n;
} while(n < 1 || n > 100);

for(sum = 0, i = 1 ; i <= n ; i += 2)
     sum += i;

cout << "The sum of the odd numbers is " << sum << endl;
```

Notes:

- In this example **`expr1`** contains two assignments; they must be separated by a comma (`,`)
- In this example **`expr3`** increments **`i`** by 2, and so **`i`** will skip all even integers.

- Consider the following code fragment:

```
int a = 0, n = 4;
for(int i = 1;i <= n;i++)
  a += 2 * i + 1;
```

- Questions:
  - What is the final value of the **`i`** variable?
  - What is the final value of the **`a`** variable?
  - What is the conceptual purpose of the **`for`** cycle?
  - Give a more descriptive name to the **`a`** variable.