

Conditional structure: if and else

The if keyword is used to execute a statement or block only if a condition is fulfilled. Its form is:
if (condition) statement

Where condition is the expression that is being evaluated. If this condition is true, statement is executed. If it is false, statement is ignored (not executed) and the program continues right after this conditional structure.

For example, the following code fragment prints x is 100 only if the value stored in the x variable is indeed 100:

```
if (x == 100)
    cout << "x is 100";
```

If we want more than a single statement to be executed in case that the condition is true we can specify a block using braces { }:

```
if (x == 100)
{
    cout << "x is ";
    cout << x;
}
```

We can additionally specify what we want to happen if the condition is not fulfilled by using the keyword else. Its form used in conjunction with if is:

```
if (condition)
    statement1
else
    statement2
```

For example:

```
if (x == 100)
    cout << "x is 100";
else
    cout << "x is not 100";
```

prints on the screen x is 100 if indeed x has a value of 100, but if it has not -and only if not- it prints out x is not 100. The if + else structures can be concatenated with the intention of verifying a range of values. The following example shows its use telling if the value currently stored in x is positive, negative or none of them (i.e. zero):

```
if (x > 0)
    cout << "x is positive";
else if (x < 0)
    cout << "x is negative";
else
    cout << "x is 0";
```

Remember that in case that we want more than a single statement to be executed, we must group them in a block by enclosing them in braces { }.

Iteration structures (loops)

Loops have as purpose to repeat a statement a certain number of times or while a condition is fulfilled.

The while loop

Its format is:

```
while (expression)
    statement
```

and its functionality is simply to repeat statement while the condition set in expression is true. For example, we are going to make a program to countdown using a while-loop:

```
// custom countdown using while
#include <iostream>
using namespace std;
int main ()
{
    int n;
    cout << "Enter the starting number > ";
    cin >> n;
    while (n>0) {
        cout << n << ", ";
        --n;
    }
    cout << "FIRE!\n";
    return 0;
}
```

When the program starts the user is prompted to insert a starting number for the countdown. Then the while loop begins, if the value entered by the user fulfills the condition $n > 0$ (that n is greater than zero) the block that follows the condition will be executed and repeated while the condition ($n > 0$) remains being true.

The whole process of the previous program can be interpreted according to the following script (beginning in main):

1. User assigns a value to n
2. The while condition is checked ($n > 0$). At this point there are two possibilities:

* condition is true: statement is executed (to step 3)

* condition is false: ignore statement and continue after it (to step 5)

3. Execute statement:

```
cout << n << ", ";
```

```
--n;
```

(prints the value of n on the screen and decreases n by 1)

4. End of block. Return automatically to step 2

5. Continue the program right after the block: print FIRE! and end the program.

When creating a while-loop, we must always consider that it has to end at some point, therefore we must provide within the block some method to force the condition to become false at some point, otherwise the loop will continue looping forever. In this case we have included `--n;` that

decreases the value of the variable that is being evaluated in the condition (n) by one - this will eventually make the condition (n>0) to become false after a certain number of loop iterations: to be more specific, when n becomes 0, that is where our while-loop and our countdown end. Of course this is such a simple action for our computer that the whole countdown is performed instantly without any practical delay between numbers.

The do-while loop

Its format is:

```
do
statement
while (condition);
```

Its functionality is exactly the same as the while loop, except that condition in the do-while loop is evaluated after the execution of statement instead of before, granting at least one execution of statement even if condition is never fulfilled. For example, the following example program echoes any number you enter until you enter 0.

```
// number echoer
#include <iostream>
using namespace std;
int main ()
{
    unsigned long n;
    do {
        cout << "Enter number (0 to end): ";
        cin >> n;
        cout << "You entered: " << n << "\n";
    } while (n != 0);
    return 0;
}
```

The do-while loop is usually used when the condition that has to determine the end of the loop is determined within the loop statement itself, like in the previous case, where the user input within the block is what is used to determine if the loop has to end. In fact if you never enter the value 0 in the previous example you can be prompted for more numbers forever.

The for loop

Its format is:

```
for (initialization; condition; increase)
```

```
statement;
```

and its main function is to repeat statement while condition remains true, like the while loop. But in addition, the for loop provides specific locations to contain an initialization statement and an increase statement. So this loop is specially designed to perform a repetitive action with a counter which is initialized and increased on each iteration.

It works in the following way:

1. Initialization is executed. Generally it is an initial value setting for a counter variable. This is executed only once.
 2. Condition is checked. If it is true the loop continues, otherwise the loop ends and statement is skipped (not executed).
 3. Statement is executed. As usual, it can be either a single statement or a block enclosed in braces { }.
 4. Finally, whatever is specified in the increase field is executed and the loop gets back to step 2.
- Here is an example of countdown using a *for* loop:

```
// countdown using a for loop
#include <iostream>
using namespace std;
int main ()
{
    for (int n=10; n>0; n--) {
        cout << n << ", ";
    }
    cout << "FIRE!\n";
    return 0;
}
```

The initialization and increase fields are optional. They can remain empty, but in all cases the semicolon signs between them must be written. For example we could write: `for (;n<10;)` if we wanted to specify no initialization and no increase; or `for (;n<10;n++)` if we wanted to include an increase field but no initialization (maybe because the variable was already initialized before).

Optionally, using the comma operator (,) we can specify more than one expression in any of the fields included in a *for* loop, like in initialization, for example. The comma operator (,) is an expression separator, it serves to a separate more than one expression where only one is generally expected. For example, suppose that we wanted to initialize more than one variable in our loop:

```
for ( n=0, i=100 ; n!=i ; n++, i-- )
{
    // whatever here...
}
```

This loop will execute for 50 times if neither n or i are modified within the loop:

n starts with a value of 0, and i with 100, the condition is $n \neq i$ (that n is not equal to i). Because n is increased by one and i decreased by one, the loop's condition will become false after the 50th loop, when both n and i will be equal to 50.

The selective structure: switch.

The syntax of the switch statement is a bit peculiar. Its objective is to check several possible constant values for an expression. Something similar to what we did at the beginning of this section with the concatenation of several if and else if instructions. Its form is the following:

```
switch (expression)
{
case constant1:
group of statements 1;
break;
case constant2:
group of statements 2;
break;
.
.
.
default:
default group of statements
}
```

It works in the following way: switch evaluates expression and checks if it is equivalent to constant1, if it is, it executes group of statements 1 until it finds the break statement. When it finds this break statement the program jumps to the end of the switch selective structure.

If expression was not equal to constant1 it will be checked against constant2. If it is equal to this, it will execute group of statements 2 until a break keyword is found, and then will jump to the end of the switch selective structure.

Finally, if the value of expression did not match any of the previously specified constants (you can include as many case labels as values you want to check), the program will execute the statements included after the default: label, if it exists (since it is optional).

Both of the following code fragments have the same behavior:

Switch example

```
switch (x) {
case 1:
cout << "x is 1";
break;
case 2:
cout << "x is 2";
break;
default:
```

```
cout << "value of x unknown";  
}
```

if-else equivalent

```
if (x == 1) {  
    cout << "x is 1";  
}  
else if (x == 2) {  
    cout << "x is 2";  
}  
else {  
    cout << "value of x unknown";  
}
```

The switch statement is a bit peculiar within the C++ language because it uses labels instead of blocks. This forces us to put break statements after the group of statements that we want to be executed for a specific condition. Otherwise the remainder statements -including those corresponding to other labels- will also be executed until the end of the switch selective block or a break statement is reached.

For example, if we did not include a break statement after the first group for case one, the program will not automatically jump to the end of the switch selective block and it would continue executing the rest of statements until it reaches either a break instruction or the end of the switch selective block. This makes unnecessary to include braces { } surrounding the statements for each of the cases, and it can also be useful to execute the same block of instructions for different possible values for the expression being evaluated. For example:

```
switch (x) {  
    case 1:  
    case 2:  
    case 3:  
        cout << "x is 1, 2 or 3";  
        break;  
    default:  
        cout << "x is not 1, 2 nor 3";  
}
```

Notice that switch can only be used to compare an expression against constants. Therefore we cannot put variables as labels (for example case n: where n is a variable) or ranges (case (1..3):) because they are not valid C++ constants. If you need to check ranges or values that are not constants, use a concatenation of if and else if statements.