Control Structures

A program is usually not limited to a linear sequence of instructions. During its process it may bifurcate, repeat code or take decisions. For that purpose, C++ provides control structures that serve to specify what has to be done by our program, when and under which circumstances.

With the introduction of control structures we are going to have to introduce a new concept: the *compound-statement* or *block*. A block is a group of statements which are separated by semicolons (;) like all C++ statements, but grouped together in a block enclosed in braces: `{ }`:

```
{ statement1; statement2; statement3; }
```

Most of the control structures that we will see in this section require a generic statement as part of its syntax. A statement can be either a simple statement (a simple instruction ending with a semicolon) or a compound statement (several instructions grouped in a block), like the one just described. In the case that we want the statement to be a simple statement, we do not need to enclose it in braces (`{}`). But in the case that we want the statement to be a compound statement it must be enclosed between braces (`{}`), forming a block.

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 `{ }`.

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	if-else equivalent

`switch (x) {`
`  case 1:`
`  cout << "x is 1";`
`  break;`
`  case 2:`
`  cout << "x is 2";`
`  break;`
`  default:`
`  cout << "value of x unknown";`
`  }`
	`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.