```
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;
}
```

```
Enter the starting number > 8
8, 7, 6, 5, 4, 3, 2, 1, FIRE!
```

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 posibilities:
   * 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 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;
}
```

```
Enter number (0 to end): 12345
You entered: 12345
Enter number (0 to end): 160277
You entered: 160277
Enter number (0 to end): 0
You entered: 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;
}
```

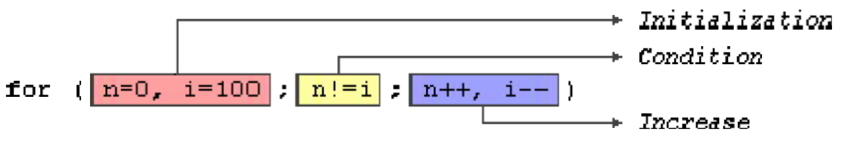
```
10, 9, 8, 7, 6, 5, 4, 3, 2, 1, FIRE!
```

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 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!=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`.

Jump statements.

The break statement

Using `break` we can leave a loop even if the condition for its end is not fulfilled. It can be used to end an infinite loop, or to force it to end before its natural end. For example, we are going to stop the count down before its natural end (maybe because of an engine check failure?):

```
// break loop example

#include <iostream>
using namespace std;

int main ()
{
  int n;
  for (n=10; n>0; n--)
  {
    cout << n << ", ";
    if (n==3)
    {
      cout << "countdown aborted!";
      break;
    }
  }
  return 0;
}
```

```
10, 9, 8, 7, 6, 5, 4, 3, countdown aborted!
```

The continue statement

The `continue` statement causes the program to skip the rest of the loop in the current iteration as if the end of the statement block had been reached, causing it to jump to the start of the following iteration. For example, we are going to skip the number 5 in our countdown:

```
// continue loop example
#include <iostream>
using namespace std;

int main ()
{
  for (int n=10; n>0; n--) {
    if (n==5) continue;
    cout << n << ", ";
  }
  cout << "FIRE!\n";
  return 0;
}
```

```
10, 9, 8, 7, 6, 4, 3, 2, 1, FIRE!
```

The goto statement

`goto` allows to make an absolute jump to another point in the program. You should use this feature with caution since its execution causes an unconditional jump ignoring any type of nesting limitations.
The destination point is identified by a label, which is then used as an argument for the goto statement. A label is made of a valid identifier followed by a colon (`:`).

Generally speaking, this instruction has no concrete use in structured or object oriented programming aside from those that low-level programming fans may find for it. For example, here is our countdown loop using `goto`:

```
// goto loop example

#include <iostream>
using namespace std;

int main ()
{
  int n=10;
  loop:
  cout << n << ", ";
  n--;
  if (n>0) goto loop;
  cout << "FIRE!\n";
  return 0;
}
```

```
10, 9, 8, 7, 6, 5, 4, 3, 2, 1, FIRE!
```

The exit function

`exit` is a function defined in the `cstdlib` library.

The purpose of `exit` is to terminate the current program with a specific exit code. Its prototype is:

```
void exit (int exitcode);
```

The `exitcode` is used by some operating systems and may be used by calling programs. By convention, an exit code of `0` means that the program finished normally and any other value means that some error or unexpected results happened.

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

Functions (I)

Using functions we can structure our programs in a more modular way, accessing all the potential that structured programming can offer to us in C++.

A function is a group of statements that is executed when it is called from some point of the program. The following is its format:

```
type name ( parameter1, parameter2, ...) { statements }
```

where:

- `type` is the data type specifier of the data returned by the function.
- `name` is the identifier by which it will be possible to call the function.
- `parameters` (as many as needed): Each parameter consists of a data type specifier followed by an identifier, like any regular variable declaration (for example: `int x`) and which acts within the function as a regular local variable. They allow to pass arguments to the function when it is called. The different parameters are separated by commas.
- `statements` is the function's body. It is a block of statements surrounded by braces `{ }`.

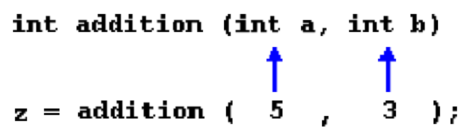
Here you have the first function example:

```cpp
// function example
#include <iostream>
using namespace std;

int addition (int a, int b)
{
  int r;
  r=a+b;
  return (r);
}

int main ()
{
  int z;
  z = addition (5,3);
  cout << "The result is " << z;
  return 0;
}
```

```
The result is 8
```

In order to examine this code, first of all remember something said at the beginning of this tutorial: a C++ program always begins its execution by the `main` function. So we will begin there.

We can see how the `main` function begins by declaring the variable `z` of type `int`. Right after that, we see a call to a function called `addition`. Paying attention we will be able to see the similarity between the structure of the call to the function and the declaration of the function itself some code lines above:



```
int addition (int a, int b)

z = addition (  5  ,  3  );
```

The parameters and arguments have a clear correspondence. Within the `main` function we called to `addition` passing two values: `5` and `3`, that correspond to the `int a` and `int b` parameters declared for function addition.

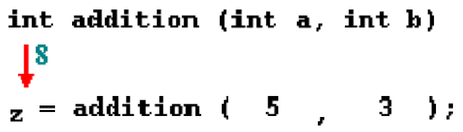
At the point at which the function is called from within `main`, the control is lost by `main` and passed to function `addition`. The value of both arguments passed in the call (`5` and `3`) are copied to the local variables `int a` and `int b` within the function.

Function `addition` declares another local variable (`int r`), and by means of the expression `r=a+b`, it assigns to `r` the result of `a` plus `b`. Because the actual parameters passed for `a` and `b` are `5` and `3` respectively, the result is `8`.

The following line of code:

```
return (r);
```

finalizes function `addition`, and returns the control back to the function that called it in the first place (in this case, `main`). At this moment the program follows it regular course from the same point at which it was interrupted by the call to `addition`. But additionally, because the `return` statement in function `addition` specified a value: the content of variable `r` (`return (r);`), which at that moment had a value of `8`. This value becomes the value of evaluating the function call.



```
int addition (int a, int b)
 8
z = addition (  5  ,   3  );
```

So being the value returned by a function the value given to the function call itself when it is evaluated, the variable `z` will be set to the value returned by `addition (5, 3)`, that is `8`. To explain it another way, you can imagine that the call to a function (`addition (5,3)`) is literally replaced by the value it returns (`8`).

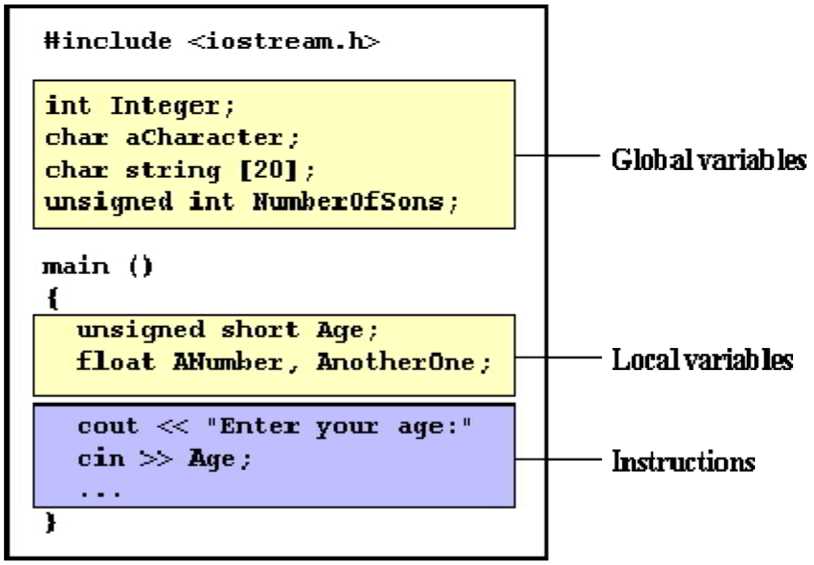
The following line of code in main is:

```
cout << "The result is " << z;
```

That, as you may already expect, produces the printing of the result on the screen.

Scope of variables

The scope of variables declared within a function or any other inner block is only their own function or their own block and cannot be used outside of them. For example, in the previous example it would have been impossible to use the variables `a`, `b` or `r` directly in function `main` since they were variables local to function `addition`. Also, it would have been impossible to use the variable `z` directly within function `addition`, since this was a variable local to the function `main`.



```
#include <iostream.h>

int Integer;
char aCharacter;
char string [20];
unsigned int NumberOfSons;

main ()
{
  unsigned short Age;
  float ANumber, AnotherOne;

  cout << "Enter your age:"
  cin >> Age;
  ...
}
```

Therefore, the scope of local variables is limited to the same block level in which they are declared. Nevertheless, we also have the possibility to declare global variables; These are visible from any point of the code, inside and outside all functions. In order to declare global variables you simply have to declare the variable outside any function or block; that means, directly in the body of the program.

And here is another example about functions:

```
// function example
#include <iostream>
using namespace std;

int subtraction (int a, int b)
{
  int r;
  r=a-b;
  return (r);
}

int main ()
{
  int x=5, y=3, z;
  z = subtraction (7,2);
  cout << "The first result is " << z << '\n';
  cout << "The second result is " << subtraction (7,2) << '\n';
  cout << "The third result is " << subtraction (x,y) << '\n';
  z= 4 + subtraction (x,y);
  cout << "The fourth result is " << z << '\n';
  return 0;
}
```

```
The first result is 5
The second result is 5
The third result is 2
The fourth result is 6
```

In this case we have created a function called `subtraction`. The only thing that this function does is to subtract both passed parameters and to return the result.

Nevertheless, if we examine function `main` we will see that we have made several calls to function `subtraction`. We have used some different calling methods so that you see other ways or moments when a function can be called.

In order to fully understand these examples you must consider once again that a call to a function could be replaced by the value that the function call itself is going to return. For example, the first case (that you should already know because it is the same pattern that we have used in previous examples):

```
z = subtraction (7,2);
cout << "The first result is " << z;
```

If we replace the function call by the value it returns (i.e., `5`), we would have:

```
z = 5;
cout << "The first result is " << z;
```

As well as

```
cout << sult second result is " << subtraction (7,2);
```

has the same result as the previous call, but in this case we made the call to `subtraction` directly as an insertion parameter for `cout`. Simply consider that the result is the same as if we had written:

```
cout << "The second result is " << 5;
```

since `5` is the value returned by `subtraction (7,2)`.

In the case of:

```
cout << sult third result is " << subtraction (x,y);
```