# Functions with no type. The use of void.

the syntax of a function declaration:

```
type  function name ( argument1, argument2 ...) statement
```

you will see that the declaration begins with a `type`, that is the type of the function itself (i.e., the type of the datum that will be returned by the function with the return statement). But what if we want to return no value?

Imagine that we want to make a function just to show a message on the screen. We do not need it to return any value. In this case we should use the **`void`** type specifier for the function. This is a special specifier that indicates absence of type.

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

void printmessage ()
{
  cout << "I'm a function!";
}

int main ()
{
  printmessage ();
  return 0;
}
```

```
I'm a function!
```

`void` can also be used in the function's parameter list to explicitly specify that we want the function to take no actual parameters when it is called. For example, function `printmessage` could have been declared as:

```
void printmessage (void)
{
  cout << "I'm a function!";
}
```

Although it is optional to specify `void` in the parameter list. In C++, a parameter list can simply be left blank if we want a function with no parameters.

What you must always remember is that the format for calling a function includes specifying its name and enclosing its parameters between parentheses. The non-existence of parameters does not exempt us from the obligation to write the parentheses. For that reason the call to `printmessage` is:

```
printmessage ();
```

The parentheses clearly indicate that this is a call to a function and not the name of a variable or some other C++ statement. The following call would have been incorrect:

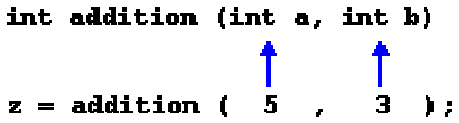
# Functions (II)

## Arguments passed by value.

Until now, in all the functions we have seen, the arguments passed to the functions have been passed *by value*. This means that when calling a function with parameters, what we have passed to the function were copies of their values but never the variables themselves. For example, suppose that we called our first function `addition` using the following code:

```
int x=5, y=3, z;
z = addition ( x , y );
```

What we did in this case was to call to function addition passing the values of `x` and `y`, i.e. `5` and `3` respectively, but not the variables `x` and `y` themselves.




This way, when the function addition is called, the value of its local variables `a` and `b` become `5` and `3` respectively, but any modification to either `a` or `b` within the function addition will not have any effect in the values of `x` and `y` outside it, because variables `x` and `y` were not themselves passed to the function, but only copies of their values at the moment the function was called.

```
// passing parameters
#include <iostream>
using namespace std;

void duplicate (int a, int b,int c)
{
  a*=2;
  b*=2;
  c*=2;
cout << "a=" << a << ", b=" << b
<<”c =” <<c ;
}

int main ()
{
  int x=1, y=3, z=7;
  duplicate (x, y, z);

  return 0;
}
```

```
a=2, b=6, c=14
```

# Default values in parameters.

When declaring a function we can specify a default value for each parameter. This value will be used if the corresponding argument is left blank when calling to the function. To do that, we simply have to use the assignment operator and a value for the arguments in the function declaration. If a value for that parameter is not passed when the function is called, the default value is used, but if a value is specified this default value is ignored and the passed value is used instead. For example:

```
// default values in functions
#include <iostream>
using namespace std;

int divide (int a, int b=2)
{
  int r;
  r=a/b;
  return (r);
}

int main ()
{
  cout << divide (12);
  cout << endl;
  cout << divide (20,4);
  return 0;
}
```

```
6
5
```

As we can see in the body of the program there are two calls to function `divide`. In the first one:

```
divide (12)
```

we have only specified one argument, but the function `divide` allows up to two. So the function `divide` has assumed that the second parameter is `2` since that is what we have specified to happen if this parameter was not passed (notice the function declaration, which finishes with `int b=2`, not just `int b`). Therefore the result of this function call is `6` (`12/2`).

In the second call:

```
divide (20,4)
```

there are two parameters, so the default value for `b` (`int b=2`) is ignored and `b` takes the value passed as argument, that is `4`, making the result returned equal to `5` (`20/4`).

# Overloaded functions.

In C++ two different functions can have the same name if their parameter types or number are different. That means that you can give the same name to more than one function if they have either a different number of parameters or different types in their parameters. For example:

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

int operate (int a, int b)
{
  return (a*b);
}

float operate (float a, float b)
{
  return (a/b);
}

int main ()
{
  int x=5,y=2;
  float n=5.0,m=2.0;
  cout << operate (x,y);
  cout << "\n";
  cout << operate (n,m);
  cout << "\n";
  return 0;
}
```

```
10
2.5
```

In this case we have defined two functions with the same name, `operate`, but one of them accepts two parameters of type `int` and the other one accepts them of type `float`. The compiler knows which one to call in each case by examining the types passed as arguments when the function is called. If it is called with two ints as its arguments it calls to the function that has two `int` parameters in its prototype and if it is called with two floats it will call to the one which has two `float` parameters in its prototype.

In the first call to `operate` the two arguments passed are of type `int`, therefore, the function with the first prototype is called; This function returns the result of multiplying both parameters. While the second call passes two arguments of type `float`, so the function with the second prototype is called. This one has a different behavior: it divides one parameter by the other. So the behavior of a call to `operate` depends on the type of the arguments passed because the function has been *overloaded*.

Notice that a function cannot be overloaded only by its return type. At least one of its parameters must have a different type