What's in a name?

Through your introductory adventures with C++ you have been using names to identify variables, functions and arrays.

There are other types of identifiers that are not part of an introductory course. We will limit ourselves to revisiting the categories mentioned above.

1. Variables

Variables as we know are used to store/retrieve values. The name of a variable has different meanings depending on how it is used:

```
int a = 7 ; //definition
a = a + 5;
```

The occurrence of *a* on the left of the assignment symbol has a different meaning from its occurrence on the right hand side.

The left *a* refers to the location in memory of variable *a* while the right *a* refers to the contents—value—of *a.*

Therefore you can view a variable as made up of a location—where in memory the variable is assigned—and a value—the contents stored in that location.

The statement *a = a +5* would thus mean: "store in the memory location bound to variable *a* the result of adding 5 to the contents of *a*."

2. Parameters

In a function, parameters represent the inputs the function needs to carry out its mission. Calling a function means using its name followed by values—arguments—consistent with the type of inputs the function expects—as declared in the prototype.

The name of the parameters is independent of the name of the variables used to call the function.

There are two different protocols to call a function: by value and by reference.

Calling by value

When a function is called by value, the value of the arguments used in the call is copied into the parameters. Given a simple function:

```
int add (int x, int l) {
  return x+s;
}
```

a call *add(s)* sends the value of *s* to the function by copying its contents to parameter *l.*

To summarize, in a call by value, the value of the arguments is copied into the corresponding function parameters. Any change in the value of *l* will not affect *s*, since *l* contains merely a copy of *s*.

Calling by reference

In a call by reference, the address of the variable is passed to the function parameter, not its value.

In C++ all parameter passing is—**by default**—by value. That is, copies are passed around.

Passing the location (address) of a name instead of a copy of its contents is referred to as passing by reference—a reference to the actual data is passed instead of its value.

This protocol is used by default with arrays. Other data types are passed by value, unless specifically marked as being passed by reference.

In order to mark a value as passed by reference, an & is used in the parameter declaration:

```
int add ( int x, int & l)
  {
    ….
   }
```

Had function *add* been defined as shown above, the call *add(5, s)* would have caused argument *s* to be passed by reference. Parameter *l* would get the address of *s* instead of the contents of *s.*

One of the effects of calling by reference is that parameters are not copies, but rather different names for the same memory location, since what is passed is the location of the original variables. Therefore, any reference parameter, if changed in the body of the function, will also change the value of the argument variable.

3. Constants

Identifiers bound to memory locations, like variables, are used, as we have seen, to store and retrieve data.

In some cases, a programmer may want to define an identifier that, once initialized, cannot be changed. That is, its value can be retrieved but not changed. In those cases the keyword **const** is used.

For example, if we needed to store time units for the purposes of a program, like the number of hours in a day, or days in a year, we could define:

```
const int HOURS = 24;
const int DAYS  = 365;
```

where the capitalization is only for easier reading, then an attempt to assign

HOURS = 12;

DAYS = 345;

would result in a compilation error, since those identifiers were defined as **const.**

As another, trivial, example:

```
const char you = 'u';
```

4. Local Variables

Variables defined inside a function cannot be shared by different functions. They are local to the function where they are defined. Variables with the same name defined in different functions are different variables, because, again, their scope is limited to those functions.

For example:

```
int fucntionA (int x)
 {
   int value;

    value = x +1 ;

return value;
}

int functionB (int z)
  {
    int value = z*2;
    return value;
}
int main ()
{
   cout << value ; //error, main cannot see variable value, it is local to the
function
    cout << z  ; //error, main cannot use variable z, local to functionB
.....

//other code
return 0;
}
```

Local variables help in defining functions that are not impacted by changes in variables that might be shared by the whole program (i.e global variables). This way, a function can carry out its computations without its variables being interfered with by any changes in a any other part of the program's code. A program can be designed in a **modular** way if functions are designed as "independent" components that interact with the whole ONLY when they are called or when they return a value.