Scope, Global Variables and Static | C++ Language Tutorial

In this lesson we are going to talk a little about **local variables** and their **scope**, **global variables** and the **static** keyword. Some other tutorials already presented these concepts earlier, most of them when talking about **variables**, but by now you already have a broader knowledge about **iteration structures**, **functions**, **blocks of statements** and even working with multiple **header and source files**. Knowing all of these, you can now better understand the concepts we are going to introduce in this lesson, and we can also go through more complex examples than we could have gone through at the time we introduced **variables**.

Scope of variables

All the **variables** in a program can be either of **local** or **global scope**. A **variable** has a **global scope** if it is declared outside of any other **function** or **block of statements**. **Global variables** can be then referred from anywhere in our code, inside every **function** or **block of statements** you wish from, as long as they were already declared. On the other hand, **local variables** can only be used from inside the **block** they were declared in, enclosed in braces (**{ }**).

For example, if a **variable** is declared at the beginning of function **main**, thus having a **local scope**, their **scope** is between their declaration point, and the point where the function ends. Note that, if you have multiple **blocks of statements** nested inside the function **main**, located after the point of declaration of that **variable**, the **variable** can referred from anywhere inside those **blocks** as well.

Because **functions** have their own **block**, **variables** declared within any **function** can only be "seen" inside that **block**, even in other nested **blocks** in the same **function**, because nested **blocks** are part of the outer **block**, and cannot be referred from any other **function**. **Variables** declared inside a **block**, may it be a **function** or any other nested **block**, are destroyed once the **block** has ended, thus can never be used again after that point.

Let's see an example, to explain things a little bit better:

```cpp
//
//  main.cpp
//  ChapterII.LocalScope
//
//  Created by Vlad Isan on 21/04/2013.
//  Copyright (c) 2013 INNERBYTE SOLUTIONS LTD. All rights reserved.
//

#include <iostream>

using namespace std;

/*
 Declaring and defining the function add.
 Inside this function we can only use the local variables:
 a, b and result, as the other variables inside function main,
 have a local scope only from inside that function,
 from the point they were declared, until the function ends.
 */
int add(int a, int b) {
    int result;

    result = a + b;

    return result;
}

int main()
{
    int x, y;

    x = 10;
    y = 5;

    cout << add(10, 5) << endl;

    /* declaring another variable y inside the block of the for iteration structure,
     which has the same name as the y variable declared at the beginning of function main.
     This is permitted, but the y variable in the outer block cannot be reffered anymore inside
     the for block, until it ends, because this local variable y in the nested block is hiding it.

     Note that i has as well a local scope, until the for block ends. After that point on, it cannot
     be reffered anymore as it is destroyed.

     Inside this block we can use any of the other variables declared prior to this block, inside function
     as long as there is no other local variable with the same name, hiding it.
     */
    for (int i = 0; i <= x; i++) {
        int y = x * i;

        cout << x << " * " << i << " = " << y << endl;
    }

    return 0;
}
```

As you can see in the previous example, which is commented throughout its execution for a better understanding, we have first declared and defined the function **add**. Note that in this **function**, we can only use the **local variables**, **a**, **b** and **result**. We can never use the **variables** declared inside the function **main**, as those are only visible inside the **block** of **main**, and in any other **nested block**, inside the **function**.

Inside the function **main** we have declared two **local variables**, **x** and **y**. Inside this **function** we also have an iteration structure, a **for loop**, in which we have also declared a **local variable** with the name of **y**, the same as the variable in the **outer block**. In this case, inside the **block of statements** of the **for loop**, we can now only refer to the **local variable y** which has been declared in this **block**, because having the same name, it now hides the other variable. As you can see we can also use the other variable**, x**, which is declared in the **outer block** and it is not hidden by any other **local variable** declared in this **nested block** with the same **name**.

The same goes for the **i** control variable we have declared in the **for** structure, it can only be referred in this **nested block**.

Global variables

Now that you've learnt what **local variables** are, it's time to learn how to use **global variables**. **Global variables** are the **variables** declared outside of any **block** and they can be used everywhere in your program, and are destroyed when the program ends.

Let's see an example of one **global variable**:

```
//
//  main.cpp
//  ChapterII.GlobalVariables
//
//  Created by Vlad Isan on 21/04/2013.
//  Copyright (c) 2013 INNERBYTE SOLUTIONS LTD. All rights reserved.
//

#include <iostream>

using namespace std;

int g_value = 0;

int main()
{
    ++g_value;
    cout << g_value << endl;

    int g_value = 100;

    cout << "g_value local variable which hides the global one: " << g_value << endl;
    cout << "g_value global variable: " << ::g_value << endl; /* In case we have a local variable with t
operator. */

    return 0;
}
```

In this example, we have declared a **global variable**, **g value**, at the beginning of our program, outside any other **block**. As you can see, we can increment it in the function **main** and the print its value to the screen.

Like with any other **variable** inside a **nested block**, we can declare another **variable** having the same name, which will hide the **global variable**. We can then access the **local variable** by its name, and as you can see it hides the **global variable**. In this case, in order to access the **global variable**, we have to use the **global scope operator**, **::**, before the **variable**'s name: **::g_value**.

I would like to point out here that you should always prevent this from happening as your code could get really ugly if you use the same name for multiple **variables** with different scopes. As you can see, we have declared our **global variable** with the **g_** prefix, just so we know, at a later time, that this is a **global variable**, and there wouldn't be the case to declare another **local variable** with the same name.

Similar to **functions**, we can also use the **global variables** in other **source files** as well, but as we learned in our lesson about forward declarations we must first let the compiler know that the variable we are using, is already declared in another place. We can use the **extern** keyword for this, which will let the compiler know that we are not declaring another **variable**, and this **variable** has already been declared in some other file.

Extern keyword

Here is an example with multiple **header** and **source files** in which we will learn how the **extern** keyword works, and how we can declare a **global variable** in **source file**, to be used in another **source file**.

globals.h

```
//
//  globals.h
//  ChapterII.ExternKeyword
//
//  Created by Vlad Isan on 21/04/2013.
//  Copyright (c) 2013 INNERBYTE SOLUTIONS LTD. All rights reserved.
//

#ifndef _globals_h_
#define _globals_h_

extern int g_value; /* Letting the compiler know this global variable is already declared
                       in a source file */

#endif /* defined(_globals_h_) */
```

Here we created a **header file** to be included in any **source file** we would like to use the **global variables** declared in it, in our case just one, **g_value**. By using the **extern** keyword, we are letting the compiler know that this **variable** we are going to use, is already declared in another **source file**.

Here is the **globals.cpp** in which we will declare this **global variable** and assign a value to it:

```
//
//  globals.cpp
//  ChapterII.ExternKeyword
//
//  Created by Vlad Isan on 21/04/2013.
//  Copyright (c) 2013 INNERBYTE SOLUTIONS LTD. All rights reserved.
//

#include "globals.h"

int g_value = 100; /* Declaring and assigning a value to the global variable.
*/
```

Nothing much to be seen in this file, we are declaring and assigning a value to the **global variable**, **g_value**, just how we would declare and assign to any other **variable**.

Let's go and have a look at the **main.cpp**:

```
//
//  main.cpp
//  ChapterII.ExternKeyword
//
//  Created by Vlad Isan on 21/04/2013.
//  Copyright (c) 2013 INNERBYTE SOLUTIONS LTD. All rights reserved.
//

#include <iostream>
#include "globals.h"
/*Including our header file which lets the compiler know we are using a global variable
declared in other source file: extern int g_value;. */

using namespace std;

int main()
{
    cout << g_value; /* Printing the global variable's value to the screen */

    return 0;
}
```

As you can see, we are including our **header file**, **globals.h**, in which we let the compiler know that we are going to use a **global variable** which is already declared in other **source file.**

After this point, we can use the **global variable**, **g_value**, just like we would use any other variable.

Now, that you know what **global variables** are, and how you can use them, you will have to make sure you will avoid them as much as you can. By using **global variables**, you are making them open to any other **source file** or **function** in your entire program. It would be a nightmare to have a very large program with lots of **source files** and **functions** that could change your **global variables**. This could lead your program to behave very strangely, due to the fact that there could be lots of places your **global variables** can be changed, and this also leads to a significant amount of time to be lost by trying to debug your program.

Static keyword

There is one more **scope** to be discussed, and one more keyword. This is the **static** keyword, that when used with a **global variable** declared outside of any **block**, will make that particular **variable** only to be accessed from the **file** in which it has been **declared**, thus having a **file scope**.

So, as with **global variables**, they can be used inside any other **function** or **block** as long as it is used in the same **file** they were declared in. One thing to note here, is that any **variable** that has been declared with the **static** keyword, cannot be used with **extern** in another **source file**.

Here is a declaration example:

```
static int fileScopeVar;
```

But, this is not its only use. This makes the **static** keyword to be one of those things people usually get confused by when programming.

Normally, as we have already learnt, **local variables** are destroyed when the **block** in which they are declared in, ends and goes out of scope. This is called **automatic storage duration**. We can also use the **auto** keyword in front of the **variable** declaration, but this would be redundant, as **local variables** are created with **automatic storage duration** by default, therefore making the **auto** keyword never really used.

By using the **static** keyword when declaring a **local variable**, we are changing the **variable** to have **static duration**, which means that the **variable**'s value will be retained even after the **block** in which it is declared in, goes out of scope and ends. **Variables** that have **static duration** can only be initialized once, and they persist throughout the entire life of your program.

Let's take the following example to help us better understand this concept:

```
//
//  main.cpp
//  ChapterII.StaticKeyword
//
//  Created by Vlad Isan on 21/04/2013.
//  Copyright (c) 2013 INNERBYTE SOLUTIONS LTD. All rights reserved.
//

#include <iostream>

using namespace std;

/*
autoIncrement function: declare a local variable
with automatic duration and initialise it with 0.

It increments its value and then prints it to the screen.
When the function ends, the variable's value is not retained
and the variable is destroyed.
 */
void autoIncrement() {
    int autoVar = 0;
    ++autoVar;

    cout << "autoVar = " << autoVar << endl;
}

/*
staticIncrement function: declare a local variable
with static duration and initialise it with 0.

It increments its value and then prints it to the screen.
When the function ends, the variable's value is retained
and the variable persists. It is only initialised once,
and the next time the function gets called, it increments the previous value
and prints it to the screen.
 */

void staticIncrement() {
    static int staticVar = 0;
    ++staticVar;

    cout << "staticVar = " << staticVar << endl;
}

int main()
{
    /*
     Call these functions five times
     and see what gets printed to the screen.
     */
    for (int i = 0; i < 5; i++) {
        autoIncrement();
        staticIncrement();
    }

    return 0;
}
```

As you can see in the previous example, we have first declared and defined a function **autoIncrement**, in which we are declaring and initializing a **local variable**, **autoVar**, with the value **0**.

We are then incrementing it and printing its value to the screen. After this **function** ends, the **variable** having **automatic storage duration**, goes out of scope and it is destroyed, making every call to this **function** to follow these steps, declaring, initializing and printing the **variable**'s value, which will always print the same value, the value of **1**.

We are also declaring and defining the function **staticIncrement**, which does the same things, the only difference being in the **local variable**'s declaration: we are declaring the **staticVar** with the **static** keyword, which makes our **local variable** to have **static duration**.

What this means, is that when the **function** ends, this **static variable** will not be destroyed. This **static variable**, **staticVar**, gets initialized only <u>once</u>, and when it goes out of scope, it is not destroyed.

What this means, is that every time we make another call to the same **function**, **staticIncrement**, the value of **staticVar** will be exactly the same value this **variable** had when the **function** ended last time we've called it. This means that, if this **variable** was previously declared and initialized in a call to the same **function**, the second time we call it, it will not declare and initialize the **static variable** again, as it persisted since the last time we've called it.

To better visualize this, go ahead and run the program in the previous example. You should get the following output:

```
autoVar = 1
staticVar = 1
autoVar = 1
staticVar = 2
autoVar = 1
staticVar = 3
autoVar = 1
staticVar = 4
autoVar = 1
staticVar = 5
```

As you can see, every time we call the **autoIncrement** function, the **local variable**, **autoVar**, will be declared and initialized again to the value **0**, as its value was not retained, and the **variable** was destroyed every time the **function** ended. Therefore, at every **function** call to **autoIncrement**, the value **1** will get printed to the screen, after we increment it once**.**

On the other hand, you can notice the value of the **static variable**, **staticVar**, was retained, and the **variable** persisted throughout the entire life of our program. The variable **staticVar** was only initialized <u>once</u>, when we first called the **staticIncrement** function. Every time we made another call to the same **function**, **staticIncrement**, the **static variable**, **staticVar**, had the value which was retained last time the **function** ended, this resulting in its value being incremented, as you can see in the output of our program.

One last thing to note about the **global** and **static variables**, is that, unlike **local variables**, if you declare and never initialize them, they will have their default values, and for the general data types, this means **0.** If you remember correctly, when we have talked about **variables**, we have learnt that if they remain uninitialized, they have an unknown value, which is not the case for **global** and **static variables**, which will mean **0** for general data types.