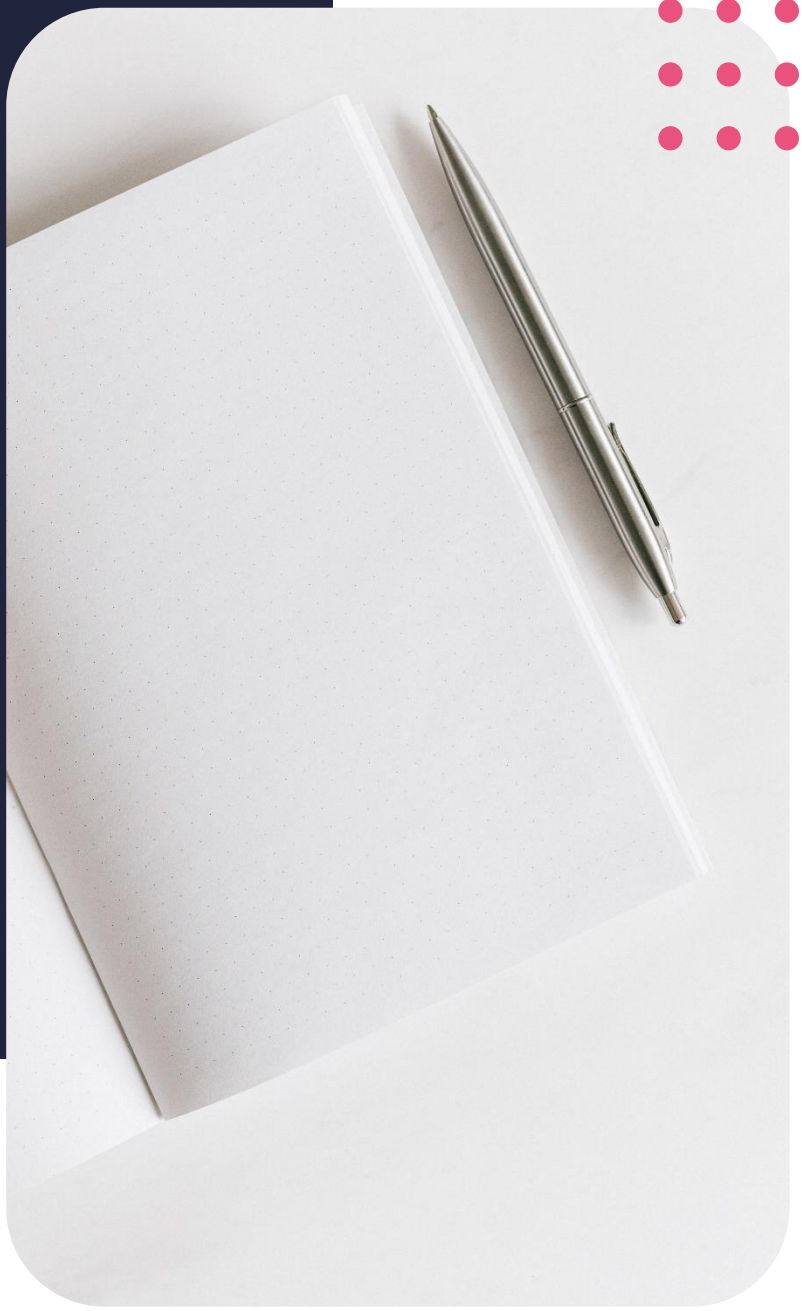
Diploma in Computer Science

Storage classes

Contents



Lesson outcomes

By the end of this lesson, you should be able to:

- Understand what scope is
- Appreciate the importance of scope in software integrity
- Recall the importance of variables as a tool for data manipulation
- Appreciate how close C is to hardware

Another look at variables

In our lesson on variables, we established that variables are memory locations in which we store data during processing. To use a variable, you need to declare it. This means that you are telling the computer to reserve a portion of memory of a certain size for the variable. You recall that it doesn't end there, as variables have types, which specify the format in which the data is stored in the variable. We then added that variables need to be initialised so that at any given time, we know the exact value that the variable holds. In this lesson, we are going to take things a bit further and specify exactly where in memory the variable is going to be stored.

Variable scope

Variable scope refers to the visibility of variables to other parts of the program. The program can either be visible globally or locally. A global variable is visible anywhere in the program. A local variable is only visible in the area where it is declared. For example, if you declare a variable within a function, the variable will only be visible within the function.

The scope of a variable is determined by the space in which the variable is contained, that is demarcated by relevant tokens. You recall that a block of code is typically contained in curly brackets. This means that the space between the open curly bracket {and the closing curly bracket} is a scope. This can be likened to your hometown. The mayor can make changes and name roads, build new structures and all. The prime minister can equally come and do the same but needs to work with the mayor to make it happen. The mayor, however, cannot go to another city and tell them what to do. Chances are nobody will even recognise him as a mayor! A global variable (the prime minister) is recognised in any part of the program (any town in the country) while the mayor (local variable) is only recognised as a mayor in his hometown. Global variables also carry their visibility among different files. By "different files", we are referring to the source files that make up the program.

Function scope

Functions, just like variables, also have scope. This is the exact reason why we put functions above the main function. Everything that is above the main function has global scope. Function scope indicates that from the beginning of the function, the entire duration of its execution to the end of the function, variables declared within it will be active. Fun fact, in C only the GOTO label has function scope.

We talked about formal parameters in our lesson on variables, referring to the parameters passed to a function at runtime. These are also a scope themselves, as they are only visible to the function.

Importance of scope

Why do we even need to hide things from each other? After all, a computer is so accurate, that we don't even need to tell it to keep things organised. A computer will only mix up things if you, the programmer mixes things up, right?

The simplest reason is we need a way to make sure that variables never ever encroach into each other's space. If by any chance, variables have the same name, scope will ensure that the program will still run correctly. Remember those rules that we looked at concerning variable scope? The local variable will always take precedence over the global variable, so the collision will not happen. Parametric values are passed from one function to another as input. If they are passed-by-reference, they are generally global in nature, which means they are accessible outside the scope of the function. If they are passed-by-value two copies of them exist, one in the calling function, and the other in the function that they are passed to.

Another reason is a function's data needs to be isolated from other functions. The variable in a limited scope when the computer gets to that point in execution. Global variables are in memory for the entire duration of the execution, which is music to the ears of attackers. It is a lot easier to look at the contents of a variable and track how it changes if it is a global variable, as there is plenty of time to analyse it. You essentially increase the attack surface, just by declaring your variable as a global variable! We mentioned parametric variables a moment ago, these can also be a point of entry for an attacker using memory exploitation hacks as passing by reference shares memory space between modules. This might sound like I'm being a big paranoid grinch, but if you look at some vulnerabilities such as rampage, you will quickly see what I'm getting at here.

Scope allows us to use memory as efficiently as possible. If you're not using a variable, it just sits there, occupying memory, waiting to be used. A local variable, on the other hand, can be destroyed as soon as we no longer need it. It is also created when the computer first encounters it, that is in its scope. This allows us to free memory when a variable is no longer being used. This also means that if you do not execute the code in a particular scope, the variables in that scope will not be created, and so the space on which those variables were supposed to sit can be used for other things.

As we saw throughout module 1, when solving a problem, we typically break down the problem into smaller pieces, then deals with the smaller problems using one or 2 functions. These functions are then put together to form a complete program that provides a solution when a user runs it. In the program, variables may sometimes be shared by all the functions. Limiting the scope allows several programmers to work on their own bit of the program to have control over what the main program will do to the variables. If all variables were global variables, it would be really hard to know when the variable would be modified, and that would make the program really unpredictable.

Storage classes

Storage classes give us granular control of the section of memory in which our variables go. This goes to some extent to define the scope of the variable in question. The storage class can be mentioned along with the variable. There are 4 types of storage classes, namely auto, register, static and extern. Each of these come with a number of fine characteristics that enable us to control our variable with much greater precision. The one thing that we are now going to focus on is the "life" of the variable, which is what storage classes define. Let's now look at each of them in greater detail.

Linkage

Linkage is the ability of an identifier, and this includes variables and functions, to be used in other scopes. If an identifier with the same name is declared in several scopes but can't be referred to in all of them, then the computer creates several copies of the same variable.

The variable can be referred to in 3 ways

No linkage: all function parameters and block scope parameters that are not extern use this linkage. This is when the variable is only, and can only be used in the scope it is in.

Internal linkage is used by static file scope identifiers. The identifier that carries this linkage can from all scopes of the current translation unit. In case you're wondering, translation unit isn't really a big fancy new term. It simply refers to a file containing source code, header files and other dependencies. These are the files that are grouped together to form an executable.

External linkage is used in non-static functions, all extern variables, and all file-scope non-static variables when can be used in any other translation units in the entire program. The only exception here is external variables that are previously declared static.

One other point to note here is that if an identifier appears with both internal and external linkage within the same translation unit, it will cause an error.

This is probably the best place to start because you're in for a surprise. For all those variables we have been creating and using so far, because we never specified a storage class, they were in the storage class auto. It's very rare to use the auto keyword, since it's the default, so your variables already carry the auto specifier. When an auto variable is declared, the variable will be saved in memory. This type if variables typically have garbage values in them, since they take on whatever's in the memory location that they're sitting on. I'm sure you remember that this is the reason why we initialise variables. The auto specifier is only allowed for variables declared at block scope. The storage is allocated when the block in which the variable was declared is entered and deallocated when the computer exits the block using any method; goto, return, reaching the end of the procedure. The only exception here is variable length arrays. For these, the memory is allocated when the declaration is executed not on block entry, and deallocated when the declaration goes out of scope, not when the block is exited. We will look at this closely in

Register

The register specifier, as the name suggests, is borrowed from computer hardware. You recall from module 1 that a register is located in the CPU and it is used by a computer to hold data and perform arithmetic and logic operations. Because of their location in relation to the CPU, and here we are referring to the actual physical location, much quicker to access a register than a memory location. This means that storing variables in registers may help to speed up your program, but here be dragons! It does not mean that you should do that willy-nilly. If you think your variable is worth being stored in a register, rather, if it is absolutely necessary to store it in a register, you can declare it with the register specifier. A good use for this specifier is when your variable is heavily used, for example, if your code loops or iterates, storing the variables in registers will increase the speed of execution, since the values are used over and over and are already stored close to the CPU. There's a bit of sad news though. Your variable is not guaranteed to be stored in a register. Your variable will be assigned to a register subject to availability. The computer can ignore the request if there aren't any available registers to accommodate your variable.

There are a few restrictions that apply to the use of the register specifier.

You cannot use pointers to reference objects that have the register storage class specifier. As we shall see later in the course, pointers refer to addresses, and a register does not have an address. The rest of the logic is pretty clear from here.

You cannot use the register storage class specifier when declaring objects in global scope.

We have just mentioned that a register does not have an address. This again, means you cannot apply the address operator (&) to a register variable.

The register storage class creates objects that have automatic block duration. This means that whenever a block is entered, storage for register objects defined in that block is accessible for use. When the block is exited, the objects are no longer accessible. If the block of code is recursive, it means that whenever the block of code is invoked, a new object will be allocated, then deallocated at the end of the block.

The register storage class creates objects that are treated as he equivalent of an object of the auto storage class, so this implies that it has no linkage, just like objects created by the auto storage class.

Static

We're now going to look at the major mechanism for enforcing information hiding in C. The static specifier can be used with functions at file scope and with variables at both file and block scope, but not in function parameter lists. Objects declared with the static storage class specifier have static storage duration, which means that memory for these objects is allocated when the program begins running and is freed when the program ends. Static storage duration for a variable is different from file or global scope: a variable can have static duration but local scope.. The static specifier can be used on data objects and anonymous unions, but you cannot use it with type declarations and function parameters. The static keyword can be used in the declaration of an array parameter to a function. The static keyword indicates that the argument passed into the function is a pointer to an array of at least the specified size. This serves to inform, the compiler that the pointer argument is never null.

Whenever you declare an object with the static storage class specifier, the object takes on internal linkage. This means each of the identifier represents the object in just one file. If an object is created in a function, the object is not destroyed when the program exits the function scope. Each time the program returns to the scope of the object, the last value carried by the object is still there.

Extern

When declaring objects that are going to be used in several source files, the extern storage class specifier is your friend. It can be used with function declarations and object declarations in file and block scope but cannot be used for function parameter lists. The extern specifier will assume a static storage duration unless combined with thread_local, which gives it thread duration. Memory is allocated for extern objects before the computer executes the main function and then deallocated when the program terminates. The scope of the variable depends on the location of the declaration in the program text. If the declaration appears within a block, the variable has block scope; otherwise, it has file scope. We are going to look at this in more detail in a bit.

An extern declaration can appear outside a function or at the beginning of a block of code. If the declaration is a function or appears outside a function is an object with external linkage, then using the keyword extern is optional.

If a declaration for an identifier already exists at file scope, any extern declaration of the same identifier found within a block points to that same object. If no other declaration for the identifier exists at file scope, the identifier will have external linkage. Also, just like the scope, the linkage of a variable using the extern storage class specifier depends on its location in the program. If it is outside a function and has been declared static earlier in the file, the object will have internal linkage, if not, then it will have external linkage. All other declarations found outside a function and don't have a storage class specifier will declare identifiers with external linkage.

To sum everything up, here's a table that shows all this seemingly complex stuff in a nutshell:

Storage Class	Declaration	Storage	Default Initial Value	Scope	Lifetime
Auto	Inside a function/block	Memory	Unpredictable	Within the function/block in which they are defined.	Within the function/block in which they are defined.
Register	Inside a function/block	CPU Registers	Garbage	Within the function/block in which they are defined.	Within the function/block in which they are defined.
Extern	Outside all functions	Memory	Zero	Entire file and other files where the variable is declared as extern. Not bound by any function, available everywhere	Program runtime

Static (local)	Inside a function/block	Memory	Zero	Within the function/block	Program runtime
Static (global)	Outside all functions	Memory	Zero	Global	Program runtime

What to use where

The whole point of learning about storage specifiers is to be able to choose the appropriate storage specifier in relation to the desired performance of your program. I've prepared a few guidelines for you to keep in mind when using storage specifiers.

Static is suitable for variables that are required to be the same every time we call it using different function calls

Register is most suitable for variables that are used frequently, like we mentioned before, when our program is running loops. You should, however, be careful, as registers a limited.

The external class should be reserved for variables that are used by all functions in the program.

If your variables don't need any of these 'special treatment' cases, then you're better off using the auto storage class.

Code security

In the modern world, it's no longer about writing code anyhow, as long as it works. There are many other factors that need to be considered, and we're going to address the elephant in the room for the first time, code security. Hacking is big business, and you probably have heard a lot of times about companies losing millions and suffering days of downtime and angry customers because of hackers. We're going to look at the very foundations of code security and some of the ways you can make sure your code is secure. Later in our course, we will look at it in a lot more detail.

Why all this?

Remember those old monitors that had a degauss button? This was used on CRT monitors to remove remnant magnetic fields that made you see ghosts and weird colours on the screen. The brain also has a similar "button", when you walk into a room and forget why you even stopped what you were doing to come to the room.

Apparently, this is a glitch in the way the brain interprets spatial information, which helps you refresh your working memory. Just like those old CRT monitors, that brain degaussing you get when you walk through a door helped our ancestors gain alertness when leaving their caves, going into the predator infested wilderness. What was once a feature is now a bug.

What we are trying to get at is that in programming, your environment matters, just like how you care to pay more attention to the environment when walking on the street than you would when having popcorn on the couch. Environment dictates where your application will reside and how it behaves. In this lesson, we explored what different scopes do and how a program behaves when stored in different

parts of memory. The location of your program during execution is very important if you want it to work right.

There are a few key questions that you need to ask yourself that help you to determine how to structure your variables and functions using scope and use storage access specifiers.

- Which operating system are you targeting?
- Are you targeting any specific hardware?

Will the application be run from a specific location or it can be run from anywhere?

Designing for a specific environment allows you to use the full potential of that environment and write extremely secure code. This is why device drivers are so efficient, the environment in which they will be used is known right off the bat. If you look around at the vulnerabilities that affect code, you will quickly realise that the same application written in the same language will have different vulnerabilities on different architectures and operating systems. That's because you may code appropriately to make your code secure, but making it portable exposes it to different environments, libraries, error handling schemes, and may even interact with users in a totally different way. You may have seen a program that has a rock-solid version on one operating system but is buggy and pretty much unusable on another. Optimising for environment may end up being a hindrance in a different OS.

Scope Interactions to Find Your Attack Surface

Before you start writing code, take a good look at your pseudocode, and even better, the flowchart. Figure out all the variables that you are going to use and try to assign the appropriate scope and storage class specifiers. This already gives you a pretty good idea of what you will type out in your Ide, and you have a far, FAR better chance of writing secure code.

It's the scope that needs to be secured, but may include components that are beyond your control, especially if your application has parts that interact with other external entities. In modern programming, this is pretty much all applications you will ever develop.

The scope is the first port of call for determining the application's attack surface, which is the unprotected or uncontrolled areas of the application which allow it to be misused or maliciously controlled. You find those areas by taking a good look at how your application interacts with outside entities.

Now you might be wondering why you are going through this trouble. Because security starts with understanding. Hackers are able to get into systems and do what do what they want by understanding how they work in much more detail than the developers who made them.

So, before you write a single line of code it's important to scope interactions. This way you know what your attack surface looks like and can address it.

Conclusion

In this lesson, we covered the scope of variables, and the type qualifiers that allows us to control the way our variables and functions behave. They help us to increase the security and stability of your program. This is a key attribute of high-quality code and can easily make or break your program. In our next lesson, we are going to look at a powerful data structure; the array, which allows us to manipulate sets of data. We will look at strings, which are in fact, arrays. We will also look at multi-dimensional arrays, which allow us to do even more with data sets. I'll save the rest for the lesson itself.

References

Anylogic.com. (2020). Help - anylogic Simulation Software. [online] Available at: https://help.anylogic.com/index.jsp?Topic=%2Fcom.anylogic.help%2Fhtml%2Fdata%2Fmodifying.html

Cppreference.com. (2011). Storage-class specifiers - cppreference.com. [online] Available at: https://en.cppreference.com/w/c/language/storage_duration

Dnews (2011). Walking Through Doorways Makes You Forget. [online] Seeker. Available at: https://www.seeker.com/walking-through-doorways-makes-you-forget-1765533917.html

Ibm.com. (2018). IBM Knowledge Center. [online] Available at: https://www.ibm.com/support/knowledgecenter/en/SSLTBW_2.2.0/com.ibm.zos.v2r2.cbclx01/regdef.htm

Itsourcecode (2020). Best C Projects with Source Code for Beginners Free Download 2020. [online] itsourcecode.com. Available at: https://itsourcecode.com/free-projects/c-projects/best-c-projects-with-source-code-for-beginners-free-download-2020/

Krill, P. (2016). 5 dev tools for better code security. [online] infoworld. Available at: https://www.infoworld.com/article/3048399/5-dev-tools-for-better-code-security.html

Poulin, C. (2014). Improve Application Security Immediately with These 5 Software Development Practices. [online] Security Intelligence. Available at: https://securityintelligence.com/improve-application-security-immediately-with-these-5-software-development-practices/

Upgrad blog. (2020). Top 7 Exciting Project ideas in C For Beginners [2020] | upgrad blog. [online] Available at: https://www.upgrad.com/blog/project-ideas-in-c-for-beginners/

Veracode. (2017). Security Starts With a Scope: Answer These Questions Before You Code | Veracode. [online] Available at: https://www.veracode.com/blog/secure-development/security-scoping-answer-these-questions-before-you-code