Abhijith CV
Senior Engineer

Understanding “static” in the C language

Source: pexels.com

“static” might seem like an easy concept to understand at first. And then you read code and you start seeing “static” popup everywhere- with not just local variables but as a storage specifier for global variables as well, and along functions and in array indices. So, what exactly does the static keyword really mean in C? Let us look at the context.

1. **As a storage class specifier for local variables.**

We are aware that all local (auto) variables are “created” and “destroyed” during every call of a function within which they are defined. Telling the compiler that a local variable is ***static*** forces the compiler to treat it differently. The static local variable is then initialized only on the first function call, and will retain its value throughout the program i.e., through multiple function calls, same as a global variable. However, unlike a global variable, auto locals still have their visibility confined within the function- same as other local variables.

How this is done is not defined in the standards, but most (if not all) implementations place static variables in the Data segment of memory instead of the Stack.

2. **As a storage class specifier for global variables.**

You may have stumbled upon code that uses **static** alongside a global variable or a function. Naturally this leads to confusion as we have always associated **static** with local variables.

A global variable or a function specified to be static only means that they are only visible within the object file it is defined in, i.e., globals and functions are limited to file scope when declared as static. Note that functions are implicitly treated as extern, this changes with static functions.

This may lead us to think that static works differently in the two contexts mentioned above. But this is simply not true.

We may think of static to do two things-

- Limit symbols to current scope
- Alter lifetime until end of program

As locals are already limited in scope by default, wrong assumptions are sometimes made that scope is only affected to static functions and globals. But a better way to think is that static still attempts to alter local variable scope, albeit redundantly.

Similarly, globals already have *static storage duration* (lifetime till end of program) and so the task of limiting scope seems redundant.

3. **Static functions**

static functions have file-scope (as opposed to the default global scope), i.e. they have internal linkage. This is useful when we want to avoid polluting the global namespace (same as with variables).

4. **As array indices in function declaration.**

```
void myfunction ( int myarray [5] );   // where foo?
```

This tells the compiler that the function *myfunction()* takes an integer array as a parameter that contains at least 5 elements. This can be useful as the compiler may optimize the resulting code or can issue warnings when violated (depending on the settings, of course).

This neat little feature was included as part of the C99 standards. Sometimes, it is more useful to add features to existing keywords than create entirely new ones. Helps with backwards compatibility and such, doesn’t it?