A Fast Review of C Essentials Part II

Structural Programming

by

Z. Cihan TAYSI

Outline

- Fixed vs. Automatic duration
- Scope
- Global variables
- The ***register*** specifier
- Storage classes
- Dynamic memory allocation

Fixed vs. Automatic Duration – I

- ***Scope*** is the technical term that denotes the region of the C source text in which a name's declaration is active.
- ***Duration*** describes the **lifetime** of a variable's memory storage.
  - Variables with ***fixed duration*** are guaranteed to retain their value even after their scope is exited.
  - There is ***no such guarantee*** for variables with ***automatic duration***.
- ***A fixed variable*** is one that is stationary, whereas ***an automatic variable*** is one whose memory storage is automatically allocated during program execution.
- Local variables (whose scope limited to a block) are automatic by default. However, you can make them fixed by using keyword static in the declaration.
- The auto keyword explicitly makes a variable automatic, but it is rarely used since it is redundant.

Fixed vs. Automatic Duration – II

```
void increment ( void ) {
   int j = 1;
   static int k = 1;
   j++;
   k++;
   printf("j : %d\t k:%d\n", j, k);
}
main ( void ) {

            increment();
            j : 2          k : 2
            increment();
            j : 2          k : 3
            increment();
            j : 2          k : 4
}
```

- Fixed variables initialized ***only once***, whereas automatic variables are initialized ***each time their block is reentered***.
- The ***increment()*** function increments two variables, ***j*** and ***k***, both initialized to 1.
  - j has automatic duration by default
  - k has fixed duration because of the ***static*** keyword

Fixed vs. Automatic Duration – III

```
void increment ( void ) {
   int j = 1;
   static int k = 1;
   j++;
   k++;
   printf("j : %d\t k:%d\n", j, k);
}
main ( void ) {

            increment();
            j : 2          k : 2
            increment();
            j : 2          k : 3
            increment();
            j : 2          k : 4
}
```

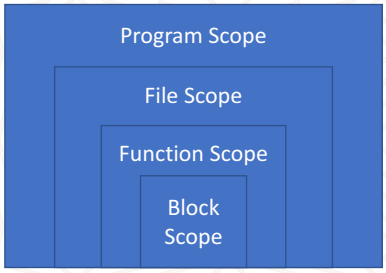
- When increment() is called the second time,
  - memory for ***j*** is reallocated and ***j*** is reinitialized to 1.
  - k has still maintained its memory address and is **NOT** reinitialized.
- Fixed variables get a default initial value of **zero**.

Scope – I

- The scope of a variable determines the region over which you can access the variable by name.
- There are four types of scope;
  - **Program scope** signifies that the variable is active among different source files that make up the entire executable program. Variables with program scope are often referred as ***global variables***.
  - **File scope** signifies that the variable is active from its declaration point to the end of the source file.
  - **Function scope** signifies that the name is active from the beginning to the end of the function.
  - **Block scope** that the variable is active from its declaration point to the end of the block which it is declared.
    - **A block is any series of statements enclosed in braces.**
    - This includes compound statements as well as function bodies.

Scope – II

```
int i ;                 // Program scope
static int j;           // File scope
func ( int k) {                 // Block scope
        int m;          // Block scope
start :                         //
Function scope
}
```



Scope – III

```
foo ( void ) {
   int j, ar[20];
   . . .
   {            // Begin debug code
      int j;    // This j does not conflict with other j's.
            for(j=0; j <= 10; ++j)
            printf("%d\t", ar[j]);
   }.           // End debug code...
    ...
}
```

- A variable with a block scope can NOT be accessed outside its block.
- It is also possible to declare a variable within a nested block.
  - can be used for debugging purposes. **see the code on the left side of the slide!**
- Although variable hiding is useful in situations such as these, it can also lead to errors that are difficult to detect!

Scope – IV

- Function scope
  - The only names that have function scope are ***goto*** labels.
  - Labels are active from the beginning to the end of a function.
    - This means that labels must be unique within a function
  - Different functions may use the same label names without creating conflicts

Scope – V

- File & Program scope
  - Giving a variable file scope makes the variable active through out the rest of the file.
    - if a file contains more than one function, all of the functions following the declaration are able to use the variable.
    - To give a variable file scope, declare it outside a function with the ***static*** keyword.
  - Variable with program scope, called global variables, are visible to routines in other files as well as their own file.
    - To create a global variable, declare it outside a function without ***static*** keyword

Global Variables

- In general, you should avoid using global variables as much as possible!
  - they make a program harder to maintain, because they increase complexity
  - create potential for conflicts between modules
  - the only advantage of global variables is that they produce faster code
- There are two types of declarations, namely, ***definition and allusion***.
- An **allusion** looks just like a definition, but instead of allocating memory for a variable, it informs the compiler that a variable of the specified type exists but is defined elsewhere.
  - extern int j;
  - The extern keyword tells the compiler that the variables are defined elsewhere.

The *register* Specifier

- The ***register*** keyword enables you to help the compiler by giving it suggestions about which variables should be kept in registers.
  - it is only a hint, not a directive, so ***compiler is free to ignore it!***
  - The behavior is implementation dependent.
- Since a variable declared with register might never be assigned a memory address, ***it is illegal to take address of a register variable.***
- A typical case to use register is when you use a counter in a loop.

```
int strlen ( register char *p)
{

        register int len=0;
        while(*p++)  {
                len++;
        }
        return len;
}
```

Storage classes summary

- **auto**
  - superfluous and rarely used.
- **static**
  - In declarations within a function, static causes variables to have fixed duration. For variables declared outside a function, the static keyword gives the variable file scope.
- **extern**
  - For variables declared within a function, it signifies a global allusion. For declarations outside of a function, extern denotes a global definition.
- **register**
  - It makes the variable automatic but also passes a hint to the compiler to store the variable in a register whenever possible.
- **const**
  - The const specifier guarantees that you can NOT change the value of the variable.
- **volatile**
  - The volatile specifier causes the compiler to turn off certain optimizations. Useful for device registers and other data segments that can change without the compiler's knowledge.