**STORAGE CLASSES**

[**VARIABLE TYPES IN C: AUTO, STATIC, REGISTER AND EXTERNAL**](https://rakanalysis.wordpress.com/2012/03/16/variable-types-in-c-auto-static-register-and-external-a-brief-guide/)

Apart from the regular types that a variable may have, like int, float, char, etc there are a set of other qualifiers which may be added onto this in order to give the variable scope of the variable, whether that is local or global. A variable may have only one of these additional qualifiers at once.

**AUTO:**

The first of these types is **auto**, and it isn’t terribly interesting. Indeed, the **auto** type is very rarely specified, as it is the default for any place where the declaration is legal and invalid elsewhere. An **auto** variable has local variable scope, meaning that it is only “visible” or can only be accessed and changed within the function or block in which it is declared. An **auto** variable disappears when it goes out of scope, freeing the memory which it was declared with, and any subsequent declaration of the variable will likely take place in a different part of memory.

The use of an **auto** variable can be illustrated:

int sum\_consecutive( int lower\_bound, int upper\_bound )

{

auto int i; /\* Value unknown! Probably something crazy. \*/

auto int sum = 0; /\* Will disappear when function finishes! \*/

for( i = lower\_bound; i <= upper\_bound; i++ )

sum += i;

return sum;

}

Neither **i** nor **sum** exist until the function is called. Both variables will probably be filled with junk until they are defined. Both variables can only be called or changed within the function in which they were called. Finally, both variables will disappear once the function has ended, their places in memory freed for other data. These variables have no persistence in memory.

Most of the time, this will be the desired effect, which is why it is the default for anything where it is legal.

**STATIC:**

Sometimes, though, you want a variable to persist throughout the running of the program. This is where the **static** type comes in. A **static** variable is declared once at the start of a program, and it doesn’t disappear once it goes out of scope, unlike an *auto* variable.

This can be illustrated by this simple function, which does little of use, but which is sufficient for demonstrative purposes:

#include<stdio.h>

int times\_called(void);

main()

{

int i;

for(i = 0; i < 10; i++)

printf( "times\_called() has been called %d times\n",

times\_called() );

}

int times\_called(void)

{

/\* The variable is declared and defined once.\*/

/\* Subsequent uses of the function will skip the declaration line. \*/

static int i = 0;

++i; /\* Incrementing the variable \*/

return i;

}

If an **auto** variable was used instead of a **static** variable in times\_called(), the return value would simply be 1 every time the function was called. The use of the **static** variable allows the variable to be incremented and for the value to be remembered throughout the program. The use of a **static** variable allows the function to work correctly. In the words of **Brian Kernighan and Dennis Ritchie (The C Programming Language, 2nd Edition, 1988)**, “ internal **static** variables provide private, permanent storage within a single function.”

if( n > 0 )

{

int i; /\* declare a new i\*/

for( i = 0; i < n; i++ )

...

}

The scope of the variable **i** is the “true” branch of the **if**; this **i** is unrelated to any **i**outside the block.

An **automatic** variable declared and initialized in a block is initialized each time the block is entered. A **static** variable is initialized only the first time the block is entered.

**REGISTER:**

A third type of variable resembles the **auto** variable declaration. This is the **register** type. A **register** declaration advises the compiler that the variable in question will be heavily used. The idea is that the **register**variables are to be placed in machine registers, which may result in smaller and faster programs. But compilers are free to ignore such advice.

The **register** declaration looks like:

register int x;

register char c;

and so on. The register declaration can only be applied to **automatic** variables and to the formal parameters of a function.

Only a few variables in each function may be kept in registers, and only certain types are allowed.

The **&** reference operator cannot be used with variables which have been declared to be **register** variables, regardless of whether they are actually placed into a **register**. The specific restrictions on number and types of **register** variables vary from machine to machine.

**EXTERN:**

The final type specifier to be covered is **extern**, but in order to understand how to use it, one must understand where it might be used. Sometimes, one doesn’t want a specific variable to be limited to one function, but instead be visible to all functions. This is where global variables come into play. A global variable can be “seen”, called and changed by all functions, and this can be illustrated here:

#include<stdio.h>

int i; /\* Global variable, declared outside any function. \*/

void funct( void );

main()

{

i = 0; /\* i isn't declared in main(), but can still be changed \*/

i++; /\* Again, changing i by incrementing it \*/

printf( "The value of i is %d\n", i );

funct();

/\* i will be different after funct() \*/

printf( "The value of i is %d\n", i );

}

void funct( void )

{

i++; /\* i isn't declared in funct() either. \*/

i += 3;

}

Running this program, we get the following results:

The value of i is 1

The value of i is 5

Even though **i** is not declared in either function, it can be accessed and changed in both functions. Notice that we don’t have to do anything special to access the variable in either function. This is because of the place where **i** was declared, right at the top of the program and before any of the functions. If we changed the place where **i** was declared, as in the following program:

#include<stdio.h>

void funct( void );

main()

{

i = 0; /\* i can't be seen by main()! \*/

i++; /\* This is a syntax error, and the compiler will complain. \*/

printf( "The value of i is %d\n", i );

funct();

/\* i will be different after funct() \*/

printf( "The value of i is %d\n", i );

}

int i; /\* This comes after main() and is no longer in its scope! \*/

void funct( void )

{

i++; /\* i isn't declared in funct(), but can be changed. \*/

i += 3;

}

We will get a compiler error. **main()** is trying to access a variable called **i**, which as far as it is concerned, doesn’t exist. However, by using the **extern** specifier, **main()** can find **i** and change it in the same way as it did in the first program illustrating external variables.

In order to do this, we change **main()** to look like this:

main()

{ /\* Declaring an external variable, which is somewhere in global scope \*/

extern int i;

i = 0; /\* i can be seen again! \*/

i++;

...

}

At this point, you may be wondering why we would use **extern** at all when we can just place the variable at the top of the source file and have it accessible to any function that needs it. The answer is that most serious projects in C are made up of several source files. A variable declared in one source file doesn’t exist in another one, and therefore, **extern** specifiers are required if one wishes to use global variables declared in other source files.

Be careful with the use of external variables. It is noted early on by **Kernighan** and **Ritchie** that the use of **external** variables can be dangerous, because they are always there even when you don’t want them. They have a tendency to make programs fragile and monolithic, and to destroy the generality and modularity of functions, such that they can’t easily be reused in other programs. This is antithetic to the aim of C to create a degree of modularity in programming, and should be avoided if at all possible.

In the absence of explicit initialization, **external** and **static** variables are guaranteed to be initialized to zero; **automatic** and **register** variables have undefined (i.e., garbage) initial values.

For **external** and **static** variables, the initialization must be done by a constant expression; the initialization is done once, conceptually before the program begins execution.

For **automatic** and **register** variables, the initialization is not restricted to being a constant; it may be any expression involving previously defined values, even function calls. The initialization is done each time the function or block is entered.

An array may be initialized by following its declaration with a list of initializers enclosed in braces and separated by commas.

int even[] = { 2, 4, 6, 8 }

when the size of the array is omitted, the compiler will compute the length by counting the initializers, of which there are 4 in this case.

If there are fewer initializers for an array than the number specified, the missing elements will be zero for **external, static,** and **automatic** variables. It is an error to have too many initializers.

Character arrays are a special case of initialization; a string may be used instead of the braces and commas notation:

char pattern[] = “computers”;

is a shorthand for the longer but equivalent

char pattern[] = { ‘c’, ‘o’, ‘m’, ‘p’, ‘u’, ‘t’, ‘e’, ‘r’, ‘s’, ‘\0’ };

In this case the size of array is five(four characters plus the terminating ‘\0’).