1.Compilation stages:

🡪Preprocessor

Compiler

Assembler

Linker

Firstly, the input file, i.e., hello.c, is passed to the preprocessor, and the preprocessor converts the

source code into expanded source code.

The extension of the expanded source code would be hello.i.

The expanded source code is passed to the compiler,

and the compiler converts this expanded source code

into assembly code. The extension of the assembly code would be hello.s.

This assembly code is then sent to the assembler, which converts the assembly code into object code.

After the creation of an object code, the linker creates the executable file. The loader will then load the executable file for the execution.

2.Memory Segments:

* Text segment
* Initialized data segment
* Uninitialized data segment
* Stack
* Heap

### 🡪1. Text/Code segment

The text segment is also known as the code segment. When we compile any program, it creates an executable file like a.out, .exe, etc., that gets stored in the text or code section of the RAM memory. If we store the instructions in the hard disk, then the speed for accessing the instructions from the hard disk becomes slower as hard disk works on the serial communication so taking the data from the hard disk will be slower, whereas the RAM is directly connected to the data and address bus so accessing the data from the RAM is faster.

2. Data section

The data which we use in our program will be stored in the data section. Since the variables declared inside the main() function are stored in the stack, but the variables declared outside the main() method will be stored in the data section. The variables declared in the data section could be stored in the form of initialized, uninitialized, and it could be local or global. Therefore, the data section is divided into four categories, i.e., initialized, uninitialized, global, or local.

Initialized Data Segment: Initialized data segment, usually called simply the Data Segment. A data segment is a portion of the virtual address space of a program, which contains the global variables and static variables that are initialized by the programmer.

Note that, the data segment is not read-only, since the values of the variables can be altered at run time.

For instance, the global string defined by char s[] = “hello world” in C and a C statement like int debug=1 outside the main (i.e. global) would be stored in the initialized read-write area. And a global C statement like const char\* string = “hello world” makes the string literal “hello world” to be stored in the initialized read-only area and the character pointer variable string in the initialized read-write area.

Ex: static int i = 10 will be stored in the data segment and global int i = 10 will also be stored in data segment

3. Uninitialized Data Segment: Uninitialized data segment often called the “bss” segment, named after an ancient assembler operator that stood for “block started by symbol.” Data in this segment is initialized by the kernel to arithmetic 0 before the program starts executing uninitialized data starts at the end of the data segment and contains all global variables and static variables that are initialized to zero or do not have explicit initialization in source code.

For instance, a variable declared static int i; would be contained in the BSS segment.

For instance, a global variable declared int j; would be contained in the BSS segment.

### 3. Stack

When we define a function and call that function then we use the stack frame. The variables which are declared inside the function are stored in the stack. The function arguments are also stored in the function as the arguments are also a part of the

function. Such a type of memory allocation is known as static memory allocation because all the variables are defined in the function, and the size of the variables is also defined at the compile time. The stack section plays a very important role in the memory because whenever the function is called, a new stack frame is created.

Stack is also used for recursive functions. When the function is called itself again and again inside the same function which causes the stack overflow condition and it leads to the segmentation fault error in the program.

### 4. Heap

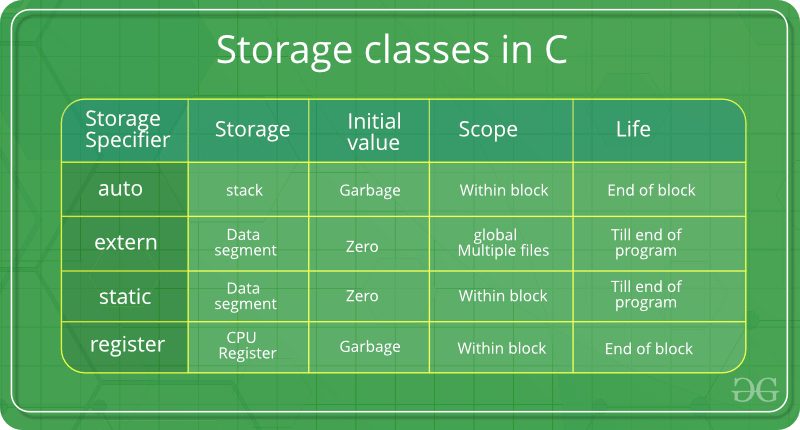
Heap memory is used for the dynamic memory allocation. Heap memory begins from the end of the uninitialized data segment and grows upwards to the higher addresses. The malloc() and calloc() functions are used to allocate the

memory in the heap. The heap memory can be used by all the shared libraries and dynamically loaded modules. The free() function is used to deallocate the memory from the heap

3.Storage Classes

🡪C language uses 4 storage classes, namely:

storage classes in c



1. auto

This is the default storage class for all the variables declared inside a function or a block. Hence, the keyword auto is rarely used while writing programs in C language. Auto variables can be only accessed within the block/function they have been declared and not outside them (which defines their scope). Of course, these can be accessed within nested blocks within the parent block/function in which the auto variable was declared.

However, they can be accessed outside their scope as well using the concept of pointers given here by pointing to the very exact memory location where the variables reside. They are assigned a garbage value by default whenever they are declared.

2. extern

Extern storage class simply tells us that the variable is defined elsewhere and not within the same block where it is used. Basically, the value is assigned to it in a different block and this can be overwritten/changed in a different block as well. So an extern variable is nothing but a global variable initialized with a legal value where it is declared in order to be used elsewhere. It can be accessed within any function/block.

Also, a normal global variable can be made extern as well by placing the ‘extern’ keyword before its declaration/definition in any function/block. This basically signifies that we are not initializing a new variable but instead, we are using/accessing the global variable only. The main purpose of using extern variables is that they can be accessed between two different files which are part of a large program.

3. static

This storage class is used to declare static variables which are popularly used while writing programs in C language. Static variables have the property of preserving their value even after they are out of their scope! Hence, static variables preserve the value of their last use in their scope. So we can say that they are initialized only once and exist till the termination of the program. Thus, no new memory is allocated because they are not re-declared.

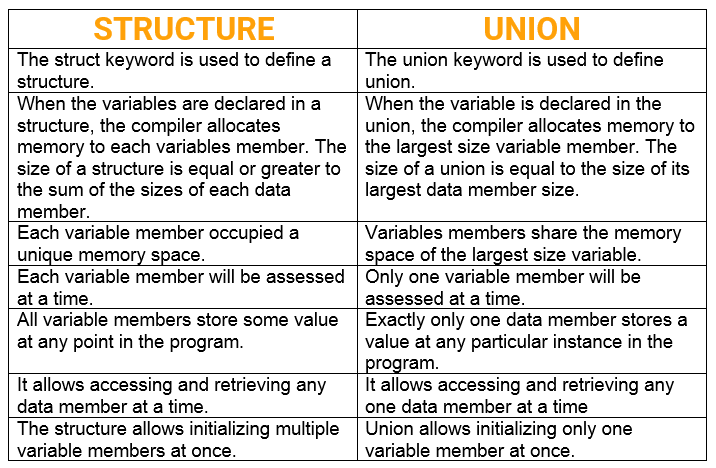
Their scope is local to the function to which they were defined. Global static variables can be accessed anywhere in the program. By default, they are assigned the value 0 by the compiler.

4. register

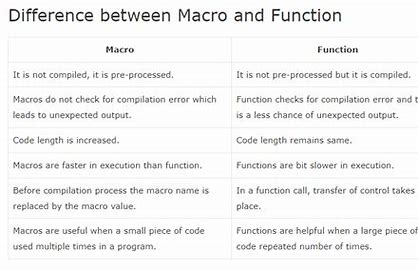
This storage class declares register variables that have the same functionality as that of the auto variables. The only difference is that the compiler tries to store these variables in the register of the microprocessor if a free register is available. This makes the use of register variables to be much faster than that of the variables stored in the memory during the runtime of the program.

If a free registration is not available, these are then stored in the memory only. Usually, a few variables which are to be accessed very frequently in a program are declared with the register keyword which improves the running time of the program. An important and interesting point to be noted here is that we cannot obtain the address of a register variable using pointers.

4.Diff b/w structure and union

🡪

5.Diff b/w function and macro



6.Diff array and pointer.

| **S.No** | **Pointer** | **Array** |
| --- | --- | --- |
| **1.** | It is declared as -:  **\*var\_name;** | It is declared as -:  **data\_type var\_name[size];** |
| **2.** | It is a variable that stores the address of another variable. | It is the collection of elements of the same data type. |
| **3.** | We can create a pointer to an array. | We can create an array of pointers. |
| **4.** | A pointer variable can store the address of only one variable at a time. | An array can store a number of elements the same size as the size of the array variable. |
| **5.** | Pointer allocation is done during runtime. | Array allocation is done during compile runtime. |
| **6.** | The nature of pointers is dynamic. The size of a pointer in C can be resized according to user requirements which means the memory can be allocated or freed at any point in time. | The nature of arrays is static. During runtime, the size of an array in C cannot be resized according to user requirements. |

7.diff b/w pointer constant vs constant to

pointer

