# Compiling a C program

Then compile it using below command.

$ gcc –Wall filename.c –o filename

The option -Wall enables all compiler’s warning messages. This option is recommended to generate better code.

## What goes inside the compilation process?

Compiler converts a C program into an executable. There are four phases for a C program to become an executable:

1. Pre-processing
2. Compilation
3. Assembly
4. Linking

By executing below command, We get the all intermediate files in the current directory along with the executable.

$gcc –Wall –save-temps filename.c –o filename

## Pre-processing

This is the first phase through which source code is passed. This phase include:

* Removal of Comments
* Expansion of Macros
* Expansion of the included files.

The preprocessed output is stored in the filename.i. Let’s see what’s inside filename.i: using $vi filename.i

**$gcc -E filename.c**

In the above output, source file is filled with lots and lots of info, but at the end our code is preserved.  
Analysis:

* printf contains now a + b rather than add(a, b) that’s because macros have expanded.
* Comments are stripped off.
* #include<stdio.h> is missing instead we see lots of code. So header files has been expanded and included in our source file.

## Compiling

**$gcc -S filename.c**

The next step is to compile filename.i and produce an; intermediate compiled output file filename.s. This file is in assembly level instructions. Let’s see through this file using $vi filename.s

## Assembly

In this phase the filename.s is taken as input and turned into filename.o by assembler. This file contain machine level instructions. At this phase, only existing code is converted into machine language, the function calls like printf() are not resolved. Let’s view this file using $objdump -S filename.o

$gcc -C filename.c

## Linking

This is the final phase in which all the linking of function calls with their definitions are done. Linker knows where all these functions are implemented. Linker does some extra work also, it adds some extra code to our program which is required when the program starts and ends. For example, there is a code which is required for setting up the environment like passing command line arguments. This task can be easily verified by using $size filename.o and $size filename. Through these commands, we know that how output file increases from an object file to an executable file. This is because of the extra code that linker adds with our program.

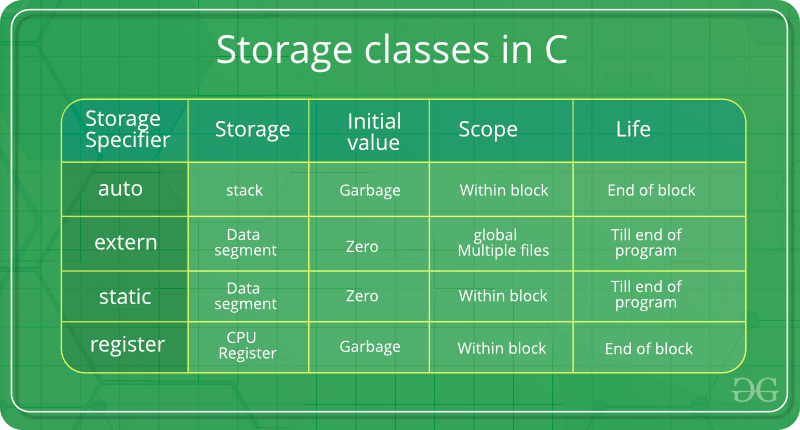
In computer science, a **linker** is a computer program that takes one or more object files generated by a compiler and combines them into one, executable program. Computer programs are usually made up of multiple modules that span separate object files, each being a compiled computer program.

**$gcc filename.c -o exec\_filename**

# **Storage Classes in C**

# Storage Classes are used to describe about the features of a variable/function. These feature basically include the scope, visibility and life-time which help us to trace the existence of a particular variable during the runtime of a program.

C language uses 4 storage classes, namely:



**Auto Storage Class**

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 resides. They are assigned a garbage value by default whenever they are declared.

The **auto** storage class is the default storage class for all local variables.

{

int mount;

auto int month;

}

The example above defines two variables with in the same storage class. 'auto' can only be used within functions, i.e., local variables.

**Extern Storage Class**

The **extern** storage class is used to give a reference of a global variable that is visible to ALL the program files. When you use 'extern', the variable cannot be initialized however, it points the variable name at a storage location that has been previously defined.

When you have multiple files and you define a global variable or function, which will also be used in other files, then *extern* will be used in another file to provide the reference of defined variable or function. Just for understanding, *extern* is used to declare a global variable or function in another file.

The extern modifier is most commonly used when there are two or more files sharing the same global variables or functions as explained below.

**First File: main.c**

#include <stdio.h>

int count ;

extern void write\_extern();

main() {

count = 5;

write\_extern();

}

**Second File: support.c**

#include <stdio.h>

extern int count;

void write\_extern(void) {

printf("count is %d\n", count);

}

Here, *extern* is being used to declare *count* in the second file, where as it has its definition in the first file, main.c. Now, compile these two files as follows −

$gcc main.c support.c

It will produce the executable program **a.out**. When this program is executed, it produces the following result −

count is 5

## The static Storage Class

The **static** storage class instructs the compiler to keep a local variable in existence during the life-time of the program instead of creating and destroying it each time it comes into and goes out of scope. Therefore, making local variables static allows them to maintain their values between function calls.

The static modifier may also be applied to global variables. Global static variables can be accessed anywhere in the program. By default, they are assigned the value 0 by the compiler.

In C programming, when **static** is used on a global variable, it causes only one copy of that member to be shared by all the objects of its class.

#include <stdio.h>

/\* function declaration \*/

void func(void);

static int count = 5; /\* global variable \*/

main() {

while(count--) {

func();

}

return 0;

}

/\* function definition \*/

void func( void ) {

static int i = 5; /\* local static variable \*/

i++;

printf("i is %d and count is %d\n", i, count);

}

When the above code is compiled and executed, it produces the following result −

i is 6 and count is 4

i is 7 and count is 3

i is 8 and count is 2

i is 9 and count is 1

i is 10 and count is 0

## The register Storage Class

This storage class declares register variables which 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 register is not available, these are then stored in the memory only. Usually 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.

The **register** storage class is used to define local variables that should be stored in a register instead of RAM. This means that the variable has a maximum size equal to the register size (usually one word) and can't have the unary '&' operator applied to it (as it does not have a memory location)

{

register int miles;

}

The register should only be used for variables that require quick access such as counters. It should also be noted that defining 'register' does not mean that the variable will be stored in a register. It means that it MIGHT be stored in a register depending on hardware and implementation restrictions.

# Structure Padding

* In order to align the data in memory, one or more empty bytes (addresses) are inserted (or left empty) between memory addresses which are allocated for other structure members while memory allocation. This concept is called structure padding.
* Architecture of a computer processor is such a way that it can read 1 word (4 byte in 32 bit processor) from memory at a time.
* To make use of this advantage of processor, data are always aligned as 4 bytes package which leads to insert empty addresses between other member’s address.
* Because of this structure padding concept in C, size of the structure is always not same as what we think.

For example, please consider below structure that has 5 members.struct student

{

int id1;

int id2;

char a;

char b;

float percentage;

};..

As per C concepts, int and float datatypes occupy 4 bytes each and char datatype occupies 1 byte for 32 bit processor. So, only 14 bytes (4+4+1+1+4) should be allocated for above structure.

* But, this is wrong. Do you know why?
* Architecture of a computer processor is such a way that it can read 1 word from memory at a time.
* 1 word is equal to 4 bytes for 32 bit processor and 8 bytes for 64 bit processor. So, 32 bit processor always reads 4 bytes at a time and 64 bit processor always reads 8 bytes at a time.
* This concept is very useful to increase the processor speed.
* To make use of this advantage, memory is arranged as a group of 4 bytes in 32 bit processor and 8 bytes in 64 bit processor.
* So, to avoid structure padding we can use pragma pack as well as an attribute.

#**pragma pack** instructs the compiler to **pack** structure members with particular alignment. Most compilers, when you declare a struct, will insert padding between members to ensure that they are aligned to appropriate addresses in memory (usually a multiple of the type's size).

|  |
| --- |
| // C program to show an example  // of Structure padding  #include <stdio.h>    struct s {      int i;      char ch;      double d;  };    int main()  {      struct s A;      printf("Size of A is: %ld", sizeof(A));  } |

**Output:**

Size of A is: 16

|  |
| --- |
| // C program to avoid structure  // padding using pragma pack  #include <stdio.h>    // To force compiler to use 1 byte packaging  #pragma pack(1)  struct s {      int i;      char ch;      double d;  };    int main()  {      struct s A;      printf("Size of A is: %ld", sizeof(A));  } |

**Output:**

Size of A is: 13

**Difference Between Structure And Union**

**Structure and union** both are user defined data types which contains variables of **different** data types. ... In **union**, the total memory space allocated is equal to the member with largest size. All other members share the same memory space. This is the biggest **difference between structure and union**.

Difference between Structure and Union

Structure Union

In structure each member get separate space in memory. Take below example.

struct student { int rollno; char gender; float marks; }s1;

The total memory required to store a structure variable is equal to the sum of size of all the members. In above case 7 bytes (2+1+4) will be required to store structure variable s1.

In union, the total memory space allocated is equal to the member with largest size. All other members share the same memory space. This is the biggest difference between structure and union.

union student { int rollno; char gender; float marks; }s1;

In above example variable marks is of float type and have largest size (4 bytes). So the total memory required to store union variable s1 is 4 bytes.

We can access any member in any sequence.

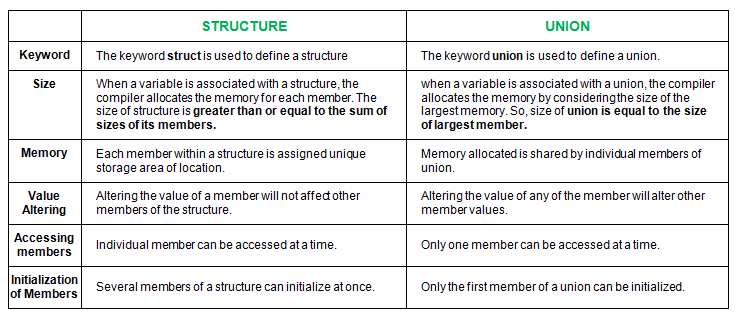
s1.rollno = 20; s1.marks = 90.0; printf("%d",s1.rollno);

The above code will work fine but will show erroneous output in the case of union. We can access only that variable whose value is recently stored.

s1.rollno = 20; s1.marks = 90.0; printf("%d",s1.rollno);

The above code will show erroneous output. The value of rollno is lost as most recently we have stored value in marks. This is because all the members share same memory space.

All the members can be initialized while declaring the variable of structure. Only first member can be initialized while declaring the variable of union. In above example we can initialize only variable rollno at the time of declaration of variable.



**Dynamic Memory Allocation**.

**Dynamic Memory Allocation** can be defined as a procedure in which the size of a data structure (like Array) is changed during the runtime.

C provides some functions to achieve these tasks. There are 4 library functions provided by C defined under **<stdlib.h>** header file to facilitate dynamic memory allocation in C programming. They are:

1. malloc()
2. calloc()
3. free()
4. realloc()

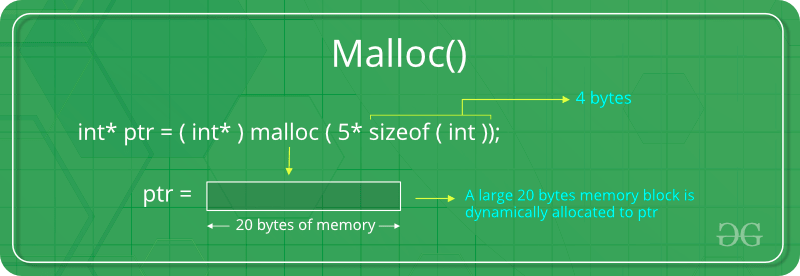
Lets see each of them in detail.

### malloc()

“**malloc”** or “**memory allocation”** method is used to dynamically allocate a single large block of memory with the specified size. It returns a pointer of type void which can be cast into a pointer of any form.If the space is insufficient, allocation fails and returns a NULL pointer

**Syntax:**

**ptr = (cast-type\*) malloc(byte-size)**

For Example:

**ptr = (int\*) malloc(100 \* sizeof(int));**

### calloc()

“**calloc”** or “**contiguous allocation”** method is used to dynamically allocate the specified number of blocks of memory of the specified type. It initializes each block with a default value ‘0’.

**Syntax:**

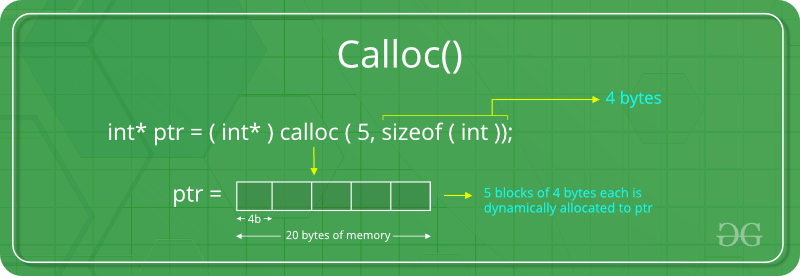
**ptr = (cast-type\*)calloc(n, element-size);**

For Example:

**ptr = (float\*) calloc(25, sizeof(float));**

This statement allocates contiguous space in memory

for 25 elements each with the size of float.



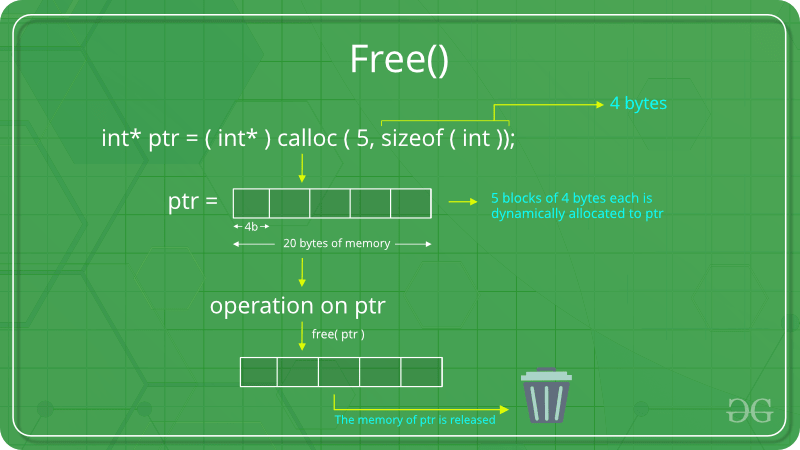
### If the space is insufficient, allocation fails and returns a NULL pointer

### free()

“**free”** method is used to dynamically **de-allocate** the memory. The memory allocated using functions malloc() and calloc() are not de-allocated on their own. Hence the free() method is used, whenever the dynamic memory allocation takes place. It helps to reduce wastage of memory by freeing it. Free function will not return anything

**Syntax:**

**free(ptr);**



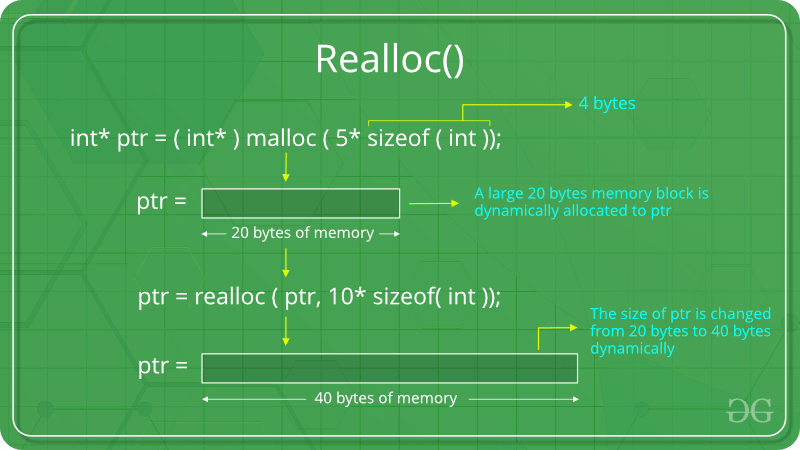
### realloc()

“**realloc”** or “**re-allocation”** method is used to dynamically change the memory allocation of a previously allocated memory. In other words, if the memory previously allocated with the help of malloc or calloc is insufficient, realloc can be used to **dynamically re-allocate memory**.

**Syntax:**

**ptr = realloc(ptr, newSize);**

where ptr is reallocated with new size 'newSize'.



If the space is insufficient, allocation fails and returns a NULL pointer.

**Function pointers :**

**Function pointers can be useful when you want to create callback mechanism, and need to pass address of an function to another function**. They can also be useful when **you** want to store an array of **functions**, to call dynamically.

In simple terms, a Callback function is a function that is not called explicitly by the programmer. Instead, there is some mechanism that continually waits for events to occur, and it will call selected functions in response to particular events.  
This mechanism is typically used when an operation(function) takes a long time for execution and the caller of the function does not want to wait till the operation is complete, but does wish to be intimated of the outcome of the operation. Typically, Callback functions help implement such an asynchronous mechanism, wherein the caller registers to get inimated about the result of the time consuming processing and continuous other operations while at a later point of time, the caller gets informed of the result.

* If you want to pass A FUNCTION as a parameter to another function, there is no other / better way to do it except function pointers.
* Function pointer can be used in place of switch case.

**Allocated Memory:**

1. when we allocated memory by using malloc() or calloc() or realloc(). first it goes make pointer is created it has some adress point when we allocate it starts from there and go on to that how much the size of the memory it want. when we dellocate it come to the starting point of it (here make pointer plays a major part in free)**.**
2. The extra space need not be just a word size - in fact, usually, it's more than that. There can be a header at the start of the allocated memory which has a size field. Simple case: Free takes the pointer passed in as argument, subtracts the size of the header struct, then reads the size field in the struct and frees the whole thing (header + allocated memory)**.**

**Most implementations of C memory allocation functions will store accounting information for each block, either in-line or separately.**

One typical way (in-line) is to actually allocate both a header and the memory you asked for, padded out to some minimum size. So for example, if you asked for 20 bytes, the system may allocate a 48-byte block:

1. 16-byte header containing size, special marker, checksum, pointers to next/previous block and so on.
2. 32 bytes data area (your 20 bytes padded out to a multiple of 16).

The address then given to you is the address of the data area. Then, when you free the block, free will simply take the address you give it and, assuming you haven't stuffed up that address or the memory around it, check the accounting information immediately before it. Graphically, that would be along the lines of:

\_\_\_\_ The allocated block \_\_\_\_

/ \

+--------+--------------------+

| Header | Your data area ... |

+--------+--------------------+

^

|

+-- The address you are given

Keep in mind the size of the header and the padding are totally implementation defined (actually, the entire thing is implementation-defined (a) but the in-line accounting option is a common one).

The checksums and special markers that exist in the accounting information are often the cause of errors like "Memory arena corrupted" or "Double free" if you overwrite them or free them twice.

The padding (to make allocation more efficient) is why you can sometimes write a little bit beyond the end of your requested space without causing problems (still, don't do that, it's undefined behaviour and, just because it works sometimes, doesn't mean it's okay to do it).

**Double pointer memory allocation:**

let int \*\*ptr;

ptr=(int \*\*)malloc(r\*sizeof(int \*));

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

ptr[i]=(int \*)malloc(c\*sizeof(int));

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

free(ptr[i]);

free(ptr);

**Diffeerent Types Of Sorting Techniques**

**Common Sorting Algorithms**

1. Merge Sort. Mergesort is a comparison-based algorithm that focuses on how to merge together two **pre**-sorted arrays such that the resulting array is also sorted.
2. Insertion Sort. ...
3. Bubble Sort. ...
4. Quicksort. ...
5. Heapsort. ...
6. **Counting** Sort.

**Bubble Sort**  
  
Bubble Sort is probably one of the oldest, most easiest, straight-forward, inefficient sorting algorithms. It works by comparing each element of the list with the element next to it and swapping them if required. With each pass, the largest of the list is "bubbled" to the end of the list whereas the smaller values sink to the bottom.  
**Selection Sort**  
  
The idea of selection sort is rather simple: we repeatedly find the next largest (or smallest) element in the array and move it to its final position in the sorted array. Assume that we wish to sort the array in increasing order, i.e. the smallest element at the beginning of the array and the largest element at the end. We begin by selecting the largest element and moving it to the highest index position. We can do this by swapping the element at the highest index and the largest element. We then reduce the effective size of the array by one element and repeat the process on the smaller (sub)array. The process stops when the effective size of the array becomes 1 (an array of 1 element is already sorted).

**Insertion Sort**  
  
The Insertion Sort algorithm is a commonly used algorithm. Even if you haven't been a programmer or a student of computer science, you may have used this algorithm. Try recalling how you sort a deck of cards. You start from the begining, traverse through the cards and as you find cards misplaced by precedence you remove them and insert them back into the right position. Eventually what you have is a sorted deck of cards. The same idea is applied in the Insertion Sort algorithm.   
**ShellSort**  
  
ShellSort is mainly a variation of Insertion Sort. In insertion sort, we move elements only one position ahead. When an element has to be moved far ahead, many movements are involved. The idea of shellSort is to allow exchange of far items. In shellSort, we make the array h-sorted for a large value of h. We keep reducing the value of h until it becomes 1. An array is said to be h-sorted if all sublists of every h’th element is sorted.  
  
  
**Heap Sort**  
  
Heap sort is a comparison based sorting technique based on Binary Heap data structure. It is similar to selection sort where we first find the maximum element and place the maximum element at the end. We repeat the same process for remaining element.  
  
  
**Merge Sort**  
  
MergeSort is a Divide and Conquer algorithm. It divides input array in two halves, calls itself for the two halves and then merges the two sorted halves.  
  
**Quick sort**  
  
Like Merge Sort, QuickSort is a Divide and Conquer algorithm. It picks an element as pivot and partitions the given array around the picked pivot. There are many different versions of quickSort that pick pivot in different ways.  
1) Always pick first element as pivot.  
2) Always pick last element as pivot (implemented below)  
3) Pick a random element as pivot.  
4) Pick median as pivot.  
  
The key process in quickSort is partition().

# Memory Layout of C Programs

A typical memory representation of C program consists of following sections.

1. Text segment  
2. Initialized data segment  
3. Uninitialized data segment  
4. Stack  
5. Heap

  
A typical memory layout of a running process

1. Text Segment:  
A text segment , also known as a code segment or simply as text, is one of the sections of a program in an object file or in memory, which contains executable instructions.

As a memory region, a text segment may be placed below the heap or stack in order to prevent heaps and stack overflows from overwriting it.

Usually, the text segment is sharable so that only a single copy needs to be in memory for frequently executed programs, such as text editors, the C compiler, the shells, and so on. Also, the text segment is often read-only, to prevent a program from accidentally modifying its instructions.

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

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

This segment can be further classified into initialized read-only area and initialized read-write area.

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 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 initialized read-only area and the character pointer variable string in initialized read-write area.

Ex: static int i = 10 will be stored in 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.

4. Stack:  
The stack area traditionally adjoined the heap area and grew the opposite direction; when the stack pointer met the heap pointer, free memory was exhausted. (With modern large address spaces and virtual memory techniques they may be placed almost anywhere, but they still typically grow opposite directions.)

The stack area contains the program stack, a LIFO structure, typically located in the higher parts of memory. On the standard PC x86 computer architecture it grows toward address zero; on some other architectures it grows the opposite direction. A “stack pointer” register tracks the top of the stack; it is adjusted each time a value is “pushed” onto the stack. The set of values pushed for one function call is termed a “stack frame”; A stack frame consists at minimum of a return address.

Stack, where automatic variables are stored, along with information that is saved each time a function is called. Each time a function is called, the address of where to return to and certain information about the caller’s environment, such as some of the machine registers, are saved on the stack. The newly called function then allocates room on the stack for its automatic and temporary variables. This is how recursive functions in C can work. Each time a recursive function calls itself, a new stack frame is used, so one set of variables doesn’t interfere with the variables from another instance of the function.

5. Heap:  
Heap is the segment where dynamic memory allocation usually takes place.

The heap area begins at the end of the BSS segment and grows to larger addresses from there.The Heap area is managed by malloc, realloc, and free, which may use the brk and sbrk system calls to adjust its size (note that the use of brk/sbrk and a single “heap area” is not required to fulfill the contract of malloc/realloc/free; they may also be implemented using mmap to reserve potentially non-contiguous regions of virtual memory into the process’ virtual address space). The Heap area is shared by all shared libraries and dynamically loaded modules in a process.

**Open and fopen**

open() is a system call and fopen() is a library function. For more details see the man page of open() and man page of fopen() and here for difference between the library and system call. The … latter link says: Well, the answer to this is the fact that fopen() is a library function which provides buffered I/O services for opening a file while open() is a system call that provides non …

fopen and its cousins are buffered. open, read, and write are not buffered.

*User buffered I/O*, shortened to *buffering* or *buffered I/O*, refers to the technique of temporarily storing the results of an I/O operation in user-space before transmitting it to the kernel (in the case of writes) or before providing it to your process (in the case of reads). By so buffering the data, you can minimize the number of system calls and can block-align I/O operations, which may improve the performance of your application.

**OSI** **Layers**

application layer

presentation layer

session layer

transport layer

network layer

datalinklayer

physical layer

# Applications of linked list data structure

A linked list is a linear data structure, in which the elements are not stored at contiguous memory locations. The elements in a linked list are linked using pointers as shown in the below image:  


Applications of linked list in computer science –

1. Implementation of [stacks](https://www.geeksforgeeks.org/stack-data-structure/) and [queues](https://www.geeksforgeeks.org/queue-data-structure/)
2. Implementation of graphs : [Adjacency list representation of graphs](https://www.geeksforgeeks.org/graph-and-its-representations/) is most popular which is uses linked list to store adjacent vertices.
3. Dynamic memory allocation : We use linked list of free blocks.
4. Maintaining directory of names
5. Performing arithmetic operations on long integers
6. Manipulation of polynomials by storing constants in the node of linked list
7. representing sparse matrices

Applications of linked list in real world-

1. Image viewer – Previous and next images are linked, hence can be accessed by next and previous button.
2. Previous and next page in web browser – We can access previous and next url searched in web browser by pressing back and next button since, they are linked as linked list.
3. Music Player – Songs in music player are linked to previous and next song. you can play songs either from starting or ending of the list.

Applications of Circular Linked Lists:

1. Useful for implementation of queue. Unlike [this](http://quiz.geeksforgeeks.org/queue-set-2-linked-list-implementation/) implementation, we don’t need to maintain two pointers for front and rear if we use circular linked list. We can maintain a pointer to the last inserted node and front can always be obtained as next of last.
2. Circular lists are useful in applications to repeatedly go around the list. For example, when multiple applications are running on a PC, it is common for the operating system to put the running applications on a list and then to cycle through them, giving each of them a slice of time to execute, and then making them wait while the CPU is given to another application. It is convenient for the operating system to use a circular list so that when it reaches the end of the list it can cycle around to the front of the list.
3. Circular Doubly Linked Lists are used for implementation of advanced data structures like [Fibonacci Heap](http://en.wikipedia.org/wiki/Fibonacci_heap).

| Fundamental Data Types | Derived Data Types |
| --- | --- |
| Fundamental data type is also called primitive data type. These are the basic data types. | Derived data type is the aggregation of fundamental data type. |
| character, integer, float, and void are fundamental data types. | Pointers, arrays, structures and unions are derived data types. |
| Character is used for characters. It can be classified as char, Signed char, Unsigned char. | Pointers are used for storing address of variables. |
| Integer is used for integers( not having decimal digits). It can be classified as signed and unsigned. Further classified as int, short int and long int. | Array is used to contain similar type of data. |
| float is used for decimal numbers. These are classified as float, double and long double. | structure is used to group items of possibly different types into a single type. |
| void is used where there is no return value required. | It is like structure but all members in union share the same memory |

**Volatile key word:**

The volatile keyword is intended to prevent the compiler from applying any optimizations on objects that can change in ways that cannot be determined by the compiler.

The declaration of a variable as volatile tells the compiler that the variable can be modified at any time externally to the implementation, for example, by the operating system, by another thread of execution such as an interrupt routine or signal handler, or by hardware. Because the value of a volatile-qualified variable can change at any time, the actual variable in memory must always be accessed whenever the variable is referenced in code. This means the compiler cannot perform optimizations on the variable, for example, caching its value in a register to avoid memory accesses.

In practice, you must declare a variable as volatile whenever you are:

* Accessing memory-mapped peripherals.
* Sharing global variables between multiple threads.
* Accessing global variables in an interrupt routine or signal handler.

The compiler does not optimize the variables you have declared as volatile.

## Proper Use of C's volatile Keyword

A variable should be declared volatile whenever its value could change unexpectedly. In practice, only three types of variables could change:

1. Memory-mapped peripheral registers

2. Global variables modified by an interrupt service routine

3. Global variables accessed by multiple tasks within a multi-threaded application