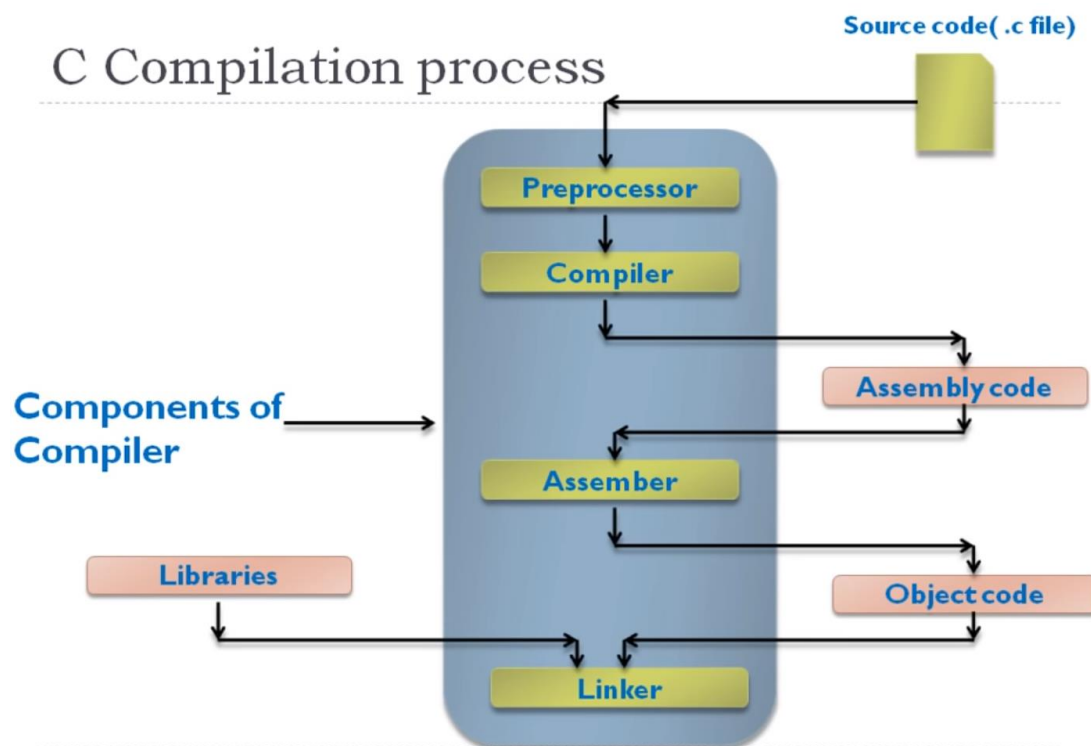


Embedded – C

1) C Program Compilation Process: -

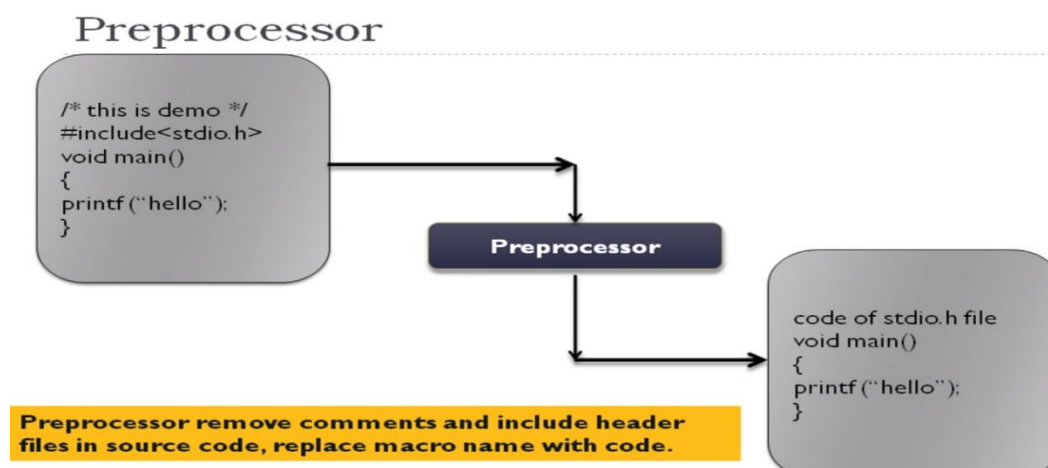
Compilation: -

Compilation is a process of converting **high level source code** into very **low-level machine code** which is different combination of binary **0** and **1**.



Preprocessor: -

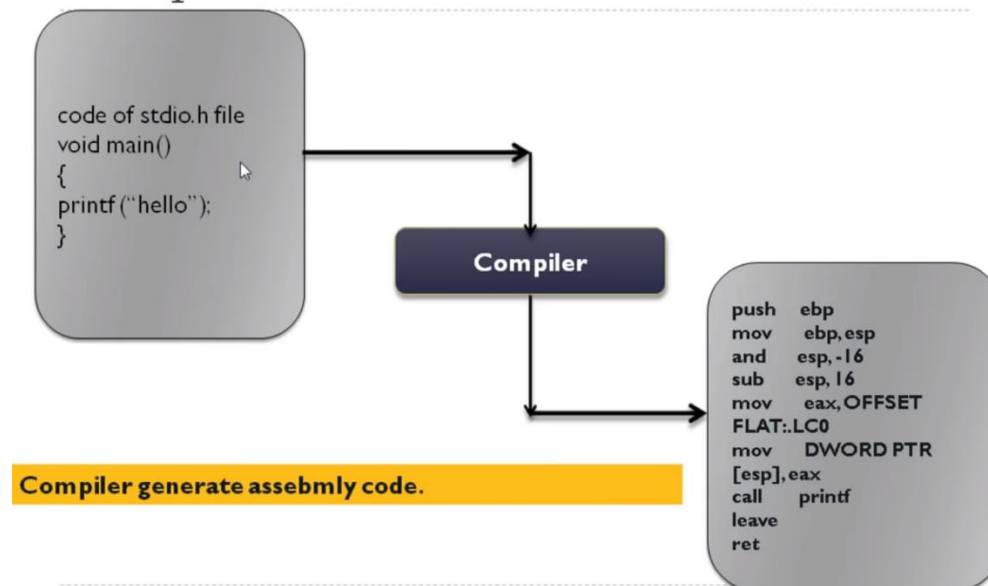
Preprocessor remove **comments** and **include header file** in source code, **replace macro name with code**. And generate (**.i file**)



Compiler: -

Compiler take .i file of preprocessor code and generate assembly code.
And generate (.s file)

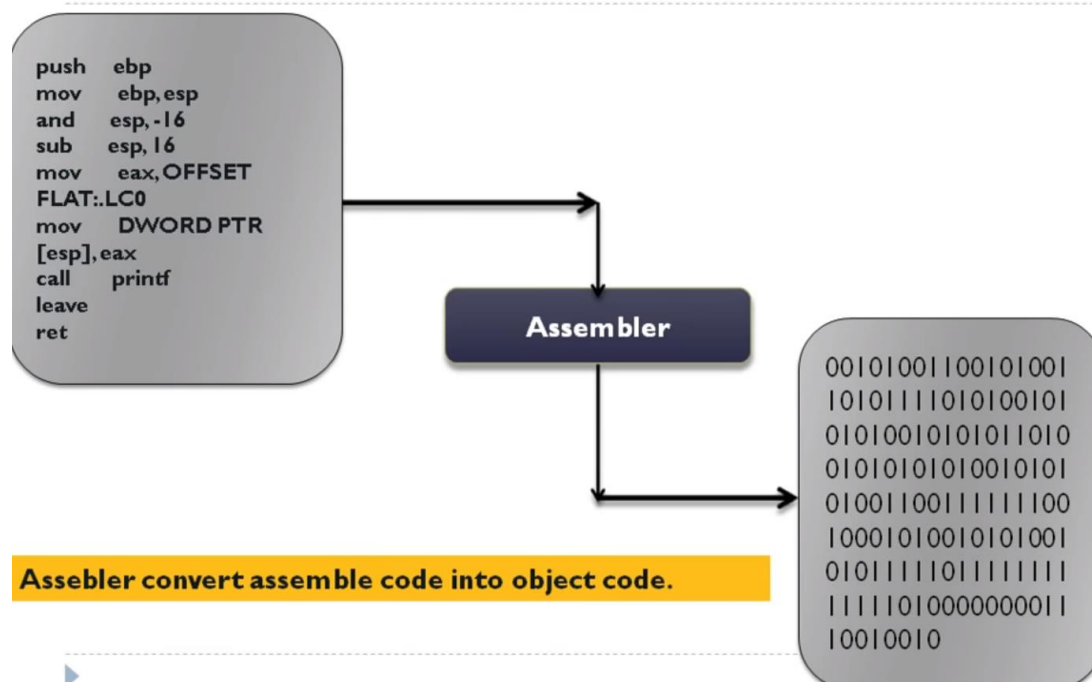
Compiler



Assembler: -

Assembler take assembly code as a input and convert into object code.
And generate (.o, .obj, file)

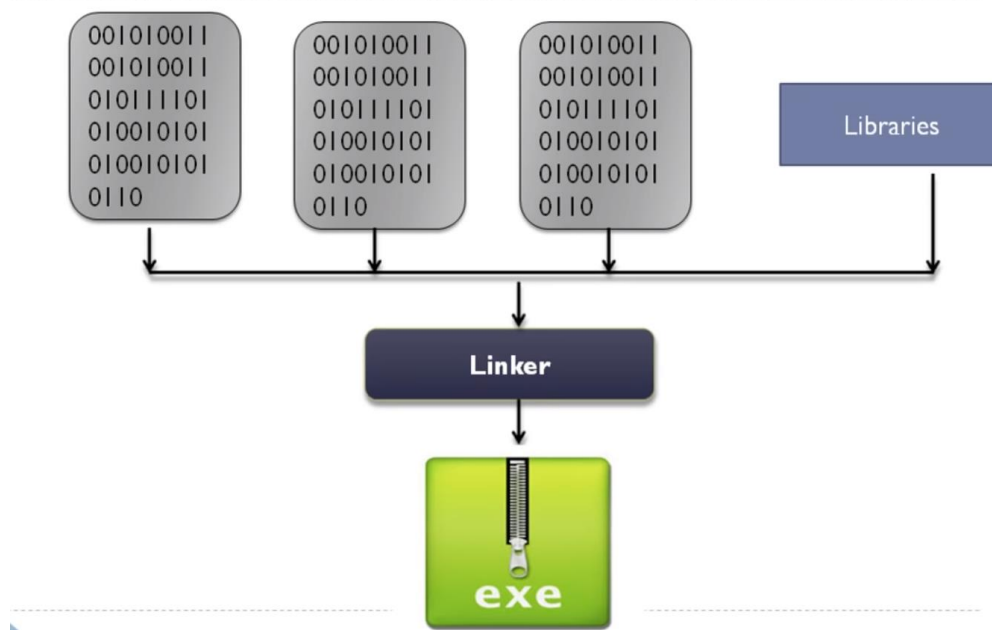
Assembler



Linker: -

Linker link all object file and library file into one executable file.
And generate (.out, .exe file)

Linker



2) Compiler Errors =

A *compiler error* indicates something that **must** be fixed before the code can be compiled.

Example: You forget a semi-colon (;) at the end of a statement and the compiler reports:
somefile.cpp:24: **parse error before `something'**

- Always remember to fix the first few errors or warnings, since they may be causing all the rest.
- Compiler messages usually list the file and line number where a problem occurs. Nonetheless, errors often occur on the lines prior to what the error message lists. Especially check the line immediately preceding where the error message indicates.
- Finally, note that some compilers may choose to call something an *error* while others may just call it a *warning* or not complain at all.

3) Compiler Warnings =

A *compiler warning* indicates you've done something bad, but not something that will prevent the code from being compiled.

You should fix whatever causes warnings since they often lead to other problems that will not be so easy to find.

Example: Your code calls the `pow()` (raise to a power) library function, but you forgot to include `math.h`.

Because you've supplied no prototype for the `pow()` function (its in `math.h`), the compiler warns you that it assumes `pow()` returns an `int` and that it assumes nothing about `pow()`'s parameters:
somefile.cpp:6: **warning:**

implicit declaration of function `int pow(...)'

this is a problem since `pow()` actually returns a `double`. In addition, the compiler can't type-check (and possibly convert) values passed to `pow()` if it doesn't know how many and what type those parameters are supposed to be.

Note: The compiler will label warnings with the word *warning* so that you can distinguish them from errors

4) **Run-Time Errors: -**

Run-time errors only occur when you *run* a program, and thus, they can only occur if there is a program to run (i.e., it must have compiled and linked without errors). **When you run the executable and something goes wrong then we call it a *run-time error*.** There are two main types of run-time errors:

1. **Fatal Errors :-**

A *fatal error* is basically when the executable crashes.

Example 1: The program divided by zero, as in:

```
int scores = 500;
int num = 0;
int avg;
avg = scores / num;
```

The program would crash saying:
Floating exception

Example 2: Segmentation faults, Bus errors.

These occur when you try to access memory that your program is not allowed to use or that doesn't exist in the computer (i.e., there is only so much memory in the computer).

Aside: Even *virtual memory* has limits.

Your program will crash giving the "**Segmentation fault**" or "**Bus error**" message.
These errors often occur due to improper use of arrays or pointers.

2. **Logic Errors :-**

A *logic error* occurs when your program simply doesn't do what you want it to.

Example: You have an infinite loop because you did not update the variable(s) used in the condition of a loop, as in:

```
cin >> account_num;
```

Assume user did not enter -1.

```
while (account_num != -1) {
    cout << "Account #: " << account_num << endl;
    ProcessAccount(account_num);
    // Oops...Forgot to read another account # here!
}
```

There are two general techniques for finding the cause of a run-time error:

- Narrow down where in the program the error occurs.
- Get more information about what is happening in the program.

Both techniques can be applied either with or without a debugging utility.

5) **Linker Error**

If you receive a **linker error**, it means that your code compiles fine, but that some **function or library** that is needed cannot be found. This occurs in what we call the *linking stage* and will prevent an executable from being generated. Many compilers do both the compiling and this linking stage.

Example 1: You misspell the name of a function (or method) when you declare, define or call it:
void Foo();

```

int main()
{
    Foo();
    return 0;
}
void foo()
{
    // do something
}

```

so that the linker complains:

`somefile.o(address): undefined reference to `Foo(void)'`
that it can't find it.

6) **Inline Function: -**

Inline Function are those function **whose definitions are small** and it can be substituted at the place where its function call is happened. **Function substitution is totally compiler choice.** If the function is inline, compiler places a copy of the code of that function at each point where the function is called at compiler time.

7) **Declaration: -**

Declaration of a variable is for informing to the compiler the following information: name of the variable, type of value it holds and the initial value if any it takes. i.e., **declaration gives details about the properties of a variable.**

Definition: -

Definition of a variable says where the variable gets stored. i.e., **memory for the variable is allocated during the definition of the variable.**

```
Int a = 4;
```

```
Int a = 5;
```

You declare variable multiple time but define it only one's

So output of this program shows error :- multiple definition of variable.

8) **Buffer overflow: -**

A buffer is a temporary area for data storage. When more data (than was originally allocated to be stored) gets placed by a program or system process, the extra data overflows. It causes some of that data to leak out into other buffers, which can corrupt or overwrite whatever data they were holding.

Stack Overflow: -

In our computer's memory, stack size is limited. If a program uses more memory space than the stack size then stack overflow will occur and can result in a program crash.

| | |
|----|--|
| 1) | If we declare large number of local variables or declare an array or matrix or any higher dimensional array of large size can result in overflow of stack. |
| 2) | If function recursively call itself infinite times then the stack is unable to store large number of local variables used by every function call and will result in overflow of stack. |

Stack is a special region of our process's memory which is used to store local variables used inside the function, parameters passed through a function and their return addresses. Whenever a new local variable is declared it is pushed onto the stack. All the variables associated with a function are

deleted and memory they use is freed up, after the function finishes running. The user does not have any need to free up stack space manually. Stack is Last-In-First-Out data structure.

Heap Overflow: -

Heap is a region of process's memory which is used to store dynamic variables. These variables are allocated using malloc() and calloc() functions and resize using realloc() function, which are inbuilt functions of C. These variables can be accessed globally and once we allocate memory on heap it is our responsibility to free that memory space after use.

There are two situations which can result in heap overflow:

| | |
|---|---|
| 1 | If we continuously allocate memory and we do not free that memory space after use it may result in memory leakage – memory is still being used but not available for other processes. |
| 2 | If we dynamically allocate large number of variables |

9) typedef:

- **typedef is used to give data type a new name**
- typedef is a keyword used in C language to assign alternative names to existing datatypes.
- Its mostly used with user defined datatypes, when names of the datatypes become slightly complicated to use in programs.
- Following is the general syntax for using typedef

```
// C program to demonstrate typedef
#include <stdio.h>

// After this line BYTE can be used
// in place of unsigned char
typedef unsigned char BYTE;

int main()
{
    BYTE b1, b2;
    b1 = 'c';
    printf("%c ", b1);
    return 0;
}
```

10) #define:

- is a C directive which is used to #define **alias**.
- In the C Programming Language, the #define directive allows the definition of macros within your source code. These macro definitions allow constant values to be declared for use throughout your code.
- Macro definitions are not variables and cannot be changed by your program code like variables. You generally use this syntax when creating constants that represent numbers, strings or expressions.

```
// C program to demonstrate #define
#include <stdio.h>

// After this line HYD is replaced by
// "Hyderabad"
#define HYD "Hyderabad"

int main()
{
    printf("%s ", HYD);
    return 0;
}
```

11) **Volatile Keyword: -**

A volatile keyword is a qualifier which prevents the objects, from the compiler optimization and tells the compiler that the value of the object can be change at any time without any action being taken by the code. It prevents from the cache a variable into a register and ensures that on every access variable is fetched from the memory.

The volatile keyword is mainly used where we directly deal with GPIO, interrupt or flag Register. It is also used where a global variable or buffer is shared between the threads.

- 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**.
- There is only one reason to use it: **When you interface with hardware**.
- Basically, C standard says that “volatile” variables can change from outside the program and that’s why compilers aren’t supposed to optimize their access.
- Objects declared as volatile are omitted from optimization because their values can be changed by code outside the scope of current code at any time.
- **>>** If an object is qualified by the volatile qualifier, the compiler reloads the value from memory each time it is accessed by the program that means it prevents from to cache a variable into a register. Reading the value from memory is the only way to check the unpredictable change of the value.
- The system always reads the current value of a volatile object from the memory location rather than keeping its value in temporary register at the point it is requested, even if a previous instruction asked for a value from the same object. So, the simple question is, how can value of a variable change in such a way that compiler cannot predict.

12) **Extern Keyword:**

External variables are also known as global variables. These variables are defined outside the function. These variables are available globally throughout the function execution. The value of global variables can be modified by the functions. “extern” keyword is used to declare and define the external variables.

Scope – They are not bound by any function. They are everywhere in the program i.e. global.

Default value – Default initialized value of global variables are Zero.

Lifetime – Till the end of the execution of the program.

Here are some important points about extern keyword in C language,

- **External variables can be declared number of times but defined only once.**
- “extern” keyword is used to extend the visibility of function or variable.
- By default, the functions are visible throughout the program, there is no need to declare or define extern functions. It just increases the redundancy.
- **Variables with “extern” keyword are only declared not defined.**
- Initialization of extern variable is considered as the definition of the extern variable.

Important:

- extern indicates that a variable is defined somewhere in the project (or **outside function block**) that you want to use. It does not allocate memory for it since you are telling the compiler that this is defined elsewhere.
- Always put the **definition of global variables**(like **float kFloat;**) in the .c file, **and put the declarations** (like **extern float kFloat;**) **in the header**.
- Otherwise when multiple .c files include the same header, there will be a **multiple definition error**.

- A variable must be defined once in one of the modules of the program. If there is no definition or more than one, an error is produced, possibly in the linking stage.
- Definition refers to the place where the variable is created or assigned storage; declaration refers to places where the nature of the variable is stated but no storage is allocated. And since it will be accessible elsewhere it needs to be static.

13) Enum =

An **enum** in c is user-defined data type and it consists a set of named constant integer. Using the enum keyword, we can declare an enumeration type with using the enumeration tag (optional) and a list of named integer.

An enumeration increases the readability of the code and easy to debug in comparison of symbolic constant (macro).

The most important thing about the enum is that it follows the scope rule and compiler automatic assign the value to its member constant.

```
#include <stdio.h>

int main(int argc, char *argv[])
{
    enum Days { Mon,Tue,Wed,Thu,Fri,Sat,Sun }; //declaration of enum in c

    enum Days eDay = Mon; //Assign Mon to enumeration variable

    printf("Mon = %d\n",eDay);

    eDay = Tue; //assign

    printf("Tue = %d\n",eDay);

    return 0;
}
```

OutPut:

Mon = 0 Tue= 1;

14) String: -

In C programming, a string is an array of characters terminated with a null character \0.

15) Array: -

An array is a group (or collection) of same data types.

In C programming, you can pass arrays to functions, however, you cannot return arrays from functions.

16) Structure: -

- Structure is a collection of different data types.
- Structure is a user defined datatype. It is used to combine different types of data into a single type.
- It can have multiple members and structure variables. The keyword "struct" is used to define structures in C language. Structure members can be accessed by using dot(.) operator.

17) Union: -

- Union is also a user defined datatype. **All the members of union share the same memory location.** Size of union is decided by the size of largest member of union.
- If you want to use same memory location for two or more members, union is the best for that.
- Unions are similar to the structure. Union variables are created in same manner as structure variables. The keyword "union" is used to define unions in C language.

| | STRUCTURE | UNION |
|----------------------------------|--|---|
| Keyword | The keyword struct is used to define a structure | The keyword union is used to define a union. |
| Size | When a variable is associated with a structure, the compiler allocates the memory for each member. The size of structure is greater than or equal to the sum of sizes of its members. | when a variable is associated with a union, the compiler allocates the memory by considering the size of the largest memory. So, size of union is equal to the size of largest member. |
| Memory | Each member within a structure is assigned unique storage area of location. | Memory allocated is shared by individual members of union. |
| Value Altering | Altering the value of a member will not affect other members of the structure. | Altering the value of any of the member will alter other member values. |
| Accessing members | Individual member can be accessed at a time. | Only one member can be accessed at a time. |
| Initialization of Members | Several members of a structure can initialize at once. | Only the first member of a union can be initialized. |

18) Structure Padding: -

Structure padding is depending on the largest member of the structure

- In the case of structure or union, the compiler inserts some extra bytes between the members of structure or union for the alignment, these extra unused bytes are called padding bytes and this technique is called padding.
- Padding increases the performance of the processor at the penalty of memory. In structure or union data members aligned as per the size of the highest bytes member to prevent the penalty of performance.

So, to avoid structure padding we can use pragma pack as well as an attribute.

Below are the solutions to avoid structure padding:

Program-1: Using pragma pack

```
// 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));
}
```

o/p = 13

Program-2: Using attribute

```
// C program to avoid structure
// padding using attribute
#include <stdio.h>

struct s {
    int i;
    char ch;
    double d;
} __attribute__((packed));
// Attribute informing compiler to pack all members

int main()
{
    struct s A;
    printf("Size of A is: %ld", sizeof(A));
}
```

o/p = 13

Remove the padding from structure use pragma pack

```
#pragma pack(push,1)
typedef struct node
{
    int a; //4 byte
    char b; //1 byte
} info_data;
#pragma pack(pop)
```

19) **Linked List:** -

Like arrays, Linked List is a linear data structure. Unlike arrays, linked list elements are not stored at a contiguous location; the elements are linked using pointers.

20) **What is pointer?**

- **The pointer variable stored in stack and can be point to stack or heap.**

21) **What is NULL Pointer?**

NULL Pointer is a pointer which is pointing to nothing. In case, if we don't have address to be assigned to a pointer, then we can simply use NULL.

At the very high level, we can think of NULL as null pointer which is used in C for various purposes. Some of the most common use cases for NULL are

a) To initialize a pointer variable when that pointer variable isn't assigned any valid memory address yet.

b) To check for null pointer before accessing any pointer variable. By doing so, we can perform error handling in pointer related code e.g. dereference pointer variable only if it's not NULL.

c) To pass a null pointer to a function argument when we don't want to pass any valid memory address.

22) Dangling pointer

A pointer pointing to a memory location that has been deleted (or freed) is called **dangling pointer**.

1. Dangling pointers arise when an object is deleted or de-allocated, without modifying the value of the pointer, so that the pointer still points to the memory location of the de-allocated memory.
2. In short pointer pointing to non-existing memory location is called dangling pointer.

23) Void pointer

When a pointer variable is declared using keyword **void** – it becomes a general purpose pointer variable. Address of any variable of any data type (char, int, float etc.) can be assigned to a void pointer variable.

Void pointer is a specific pointer type – **void *** – a pointer that points to some data location in storage, which doesn't have any specific type. Void refers to the type. Basically the type of data that it points to is can be any. If we assign address of char data type to void pointer it will become char Pointer, if int data type then int pointer and so on. Any pointer type is convertible to a void pointer hence it can point to any value.

Important Points

1. void pointers **cannot be dereferenced**. It can however be done using typecasting the void pointer
2. Pointer arithmetic is not possible on pointers of void due to lack of concrete value and thus size.

24) Wild Pointer

A pointer which has not been initialized to anything (not even NULL) is known as **wild pointer**. The pointer may be initialized to a non-NULL garbage value that may not be a valid address.

Ex. `Int *p` #pointer is uninitialized which is a wild pointer

```
int x=10;
```

```
Int *p=&x; #this is a valid pointer.
```

25) Constant and pointer Relation

1. `const char *ptr =`

This is a pointer to a constant character. You cannot change the value pointed by ptr, but you can change the pointer itself. "`const char *`" is a (non-const) pointer to a const char.

```

1 // C program to illustrate
2 // char const *p
3 #include<stdio.h>
4 #include<stdlib.h>
5
6 int main()
7 {
8     char a = 'A', b = 'B';
9     const char *ptr = &a;
10
11     /*ptr = b; illegal statement
12     //(assignment of read-only location *ptr)
13
14     // ptr can be changed
15     printf( "value pointed to by ptr: %c\n", *ptr);
16     ptr = &b;
17     printf( "value pointed to by ptr: %c\n", *ptr);
18 }
19 |

```

2. `char *const ptr =`

`char *const ptr` : This is a constant pointer to non-constant character. You cannot change the pointer `p`, but can change the value pointed by `ptr`.

```

1 // C program to illustrate
2 // char* const p
3 #include<stdio.h>
4 #include<stdlib.h>
5
6 int main()
7 {
8     char a = 'A', b = 'B';
9     char *const ptr = &a;
10     printf( "Value pointed to by ptr: %c\n", *ptr);
11     printf( "Address ptr is pointing to: %d\n\n", ptr);
12
13     //ptr = &b; illegal statement
14     //(assignment of read-only variable ptr)
15
16     // changing the value at the address
17     //ptr is pointing to
18     *ptr = b;
19     printf( "Value pointed to by ptr: %c\n", *ptr);
20     printf( "Address ptr is pointing to: %d\n", ptr);
21 }
22

```

3. `const char *const ptr =`

`const char *const ptr` : This is a constant pointer to constant character. You can neither change the value pointed by `ptr` nor the pointer `ptr`.

```

1 // C program to illustrate
2 //const char * const ptr
3 #include<stdio.h>
4 #include<stdlib.h>
5
6 int main()
7 {
8     char a = 'A', b = 'B';
9     const char *const ptr = &a;
10
11     printf( "Value pointed to by ptr: %c\n", *ptr);
12     printf( "Address ptr is pointing to: %d\n\n", ptr);
13
14     // ptr = &b; illegal statement
15     //(assignment of read-only variable ptr)
16     // *ptr = b; illegal statement
17     //(assignment of read-only location *ptr)
18
19 }
20

```

26) Referencing: -

Referencing means taking the address of an existing variable (using &) to set a pointer variable. In order to be valid, a pointer has to be set to the address of a variable of the same type as the pointer, without the asterisk:

```

Int a = 12;
Int *p;
P = &a; >> this is referencing of pointer

```

27) De-Referencing: -

Dereferencing a pointer means using the * operator (asterisk character) to retrieve the value from the memory address that is pointed by the pointer: NOTE: The value stored at the address of the pointer must be a value OF THE SAME TYPE as the type of variable the pointer "points" to, but there is no guarantee this is the case unless the pointer was set correctly. The type of variable the pointer points to is the type less the outermost asterisk.

```

Int a;
a = *p; >> this is de-referencing of pointer

```

28) Function Pointer

It is similar to the other **pointers** but the only difference is that it points to a function instead of the variable.

In the other word, we can say, a function pointer is a type of pointer that store the address of a function and these pointed functions can be invoked by function pointer in a program whenever required.

Example: -

1. I Created a function pointer and initialize to NULL.
It a integer function pointer it return value is integer you can use any return

Data type.

```
Int (*func_ptr) (int,int,char,float) = NULL
```

```
Func_ptr = add_two_num //function address
```

2) Below is a example of type define function pointer

```
Typedef int (*func_ptr) (int, int);
```

```
Func_ptr new_func_ptr;
```

29) **Memory leak: -**

- Memory leak occurs when programmers create a memory in heap and forget to delete it.
- To avoid memory leaks, memory allocated on heap should always be freed when no longer needed.

30) **pointers advantages and disadvantages**

Advantages: -

- Pointers provide direct access to memory
- Pointers provide a way to return more than one value to the functions
- Reduces the storage space and complexity of the program
- Reduces the execution time of the program
- Provides an alternate way to access array elements
- Pointers can be used to pass information back and forth between the calling function and called function.
- Pointers allows us to perform dynamic memory allocation and deallocation.
- Pointers helps us to build complex data structures like linked list, stack, queues, trees, graphs etc.
- Pointers allows us to resize the dynamically allocated memory block.
- Addresses of objects can be extracted using pointers

Disadvantages: -

- Uninitialized pointers might cause segmentation fault.
- Dynamically allocated block needs to be freed explicitly. Otherwise, it would lead to memory leak.
- Pointers are slower than normal variables.
- If pointers are updated with incorrect values, it might lead to memory corruption.
- Basically, pointer bugs are difficult to debug. Its programmer's responsibility to use pointers effectively and correctly.

31) **Core Dump/Segmentation fault** is a specific kind of error caused by accessing memory that **"does not belong to you."**

When a piece of code tries to do read and write operation in a read only location in memory or freed block of memory, it is known as core dump.

It is an error indicating memory corruption.


32) **Dynamic Memory Allocation**

1. 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.

Malloc()

```
int* ptr = (int*) malloc ( 5* sizeof ( int ));
```

ptr =  ← 20 bytes of memory →

4 bytes


A large 20 bytes memory block is dynamically allocated to ptr

2. 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’.

Calloc()

```
int* ptr = (int*) calloc ( 5, sizeof ( int ));
```

ptr =  ← 4b →

← 20 bytes of memory →

4 bytes


5 blocks of 4 bytes each is dynamically allocated to ptr

3. 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.

Realloc()

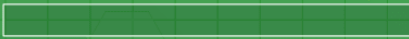
```
int* ptr = (int*) malloc ( 5* sizeof ( int ));
```

ptr =  ← 20 bytes of memory →

4 bytes

A large 20 bytes memory block is dynamically allocated to ptr

```
ptr = realloc ( ptr, 10* sizeof ( int ));
```

ptr =  ← 40 bytes of memory →


The size of ptr is changed from 20 bytes to 40 bytes dynamically

4. 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()

```
int* ptr = (int*) calloc ( 5, sizeof ( int ));
```

ptr =  ← 4b →


← 20 bytes of memory →

4 bytes

5 blocks of 4 bytes each is dynamically allocated to ptr

operation on ptr

free(ptr)

 ← 20 bytes of memory →

The memory of ptr is released

33) **Storage Classes in C: -**

Each variable has a storage class which defines the features of that variable. It tells the compiler about where to store the variable, its initial value, scope (visibility level) and lifetime (global or local).

There are four storage classes in C

| Storage classes in C | | | | |
|----------------------|--------------|---------------|--------------------------|---------------------|
| Storage Specifier | Storage | Initial value | Scope | Life |
| auto | stack | Garbage | Within block | End of block |
| extern | Data segment | Zero | global Multiple files | Till end of program |
| static | Data segment | Zero | Within block | Till end of program |
| register | CPU Register | Garbage | Within block | End of block |

1. **Automatic 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.

2. **External storage class**

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. For more information on how extern variables work, have a look at this link.

3. Static storage class

Value of the variable persist between different function call.

This storage class is used to declare static variables which are popularly used while writing programs in C language. Static variables have a 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 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.**

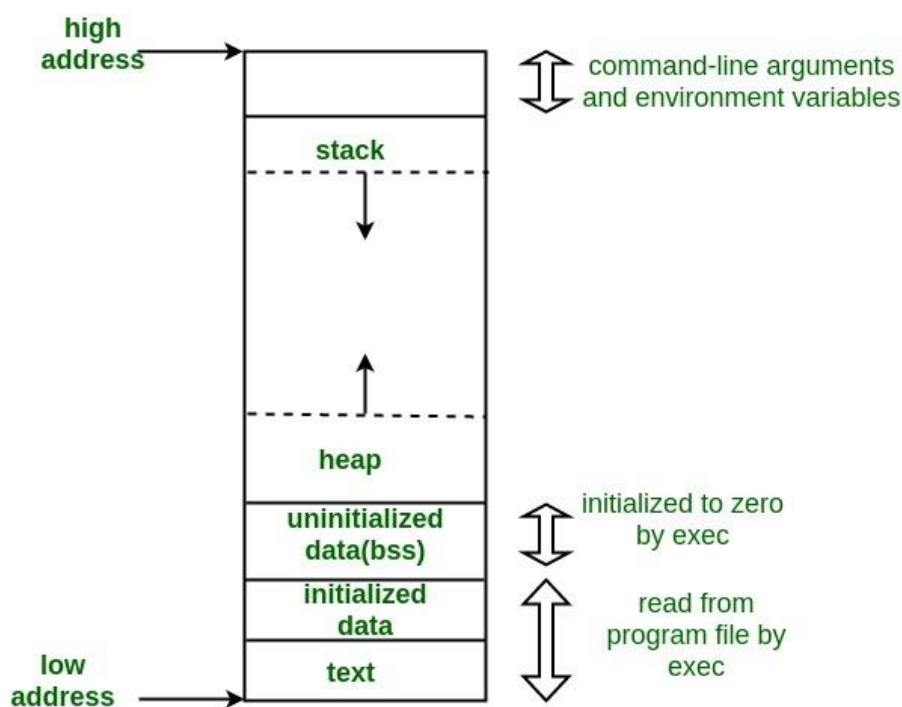
34) Static Vs. Dynamic Memory allocation

DIFFERENCE BETWEEN STATIC MEMORY ALLOCATION AND DYNAMIC MEMORY ALLOCATION IN C:

| Static memory allocation | Dynamic memory allocation |
|--|--|
| In static memory allocation, memory is allocated while writing the C program. Actually, user requested memory will be allocated at compile time. | In dynamic memory allocation, memory is allocated while executing the program. That means at run time. |
| Memory size can't be modified while execution. Example: array | Memory size can be modified while execution. Example: Linked list |

In static memory allocation memory allocated to variable during compile time.

35) Memory Layout of c program



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

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.

36) Static vs. Global: -

The global variable has global scope, I mean it can be accessed by any function, from any file, whereas **static variable has file scope**, it is not possible to access the variable from any other file. This technique is helpful when u want to make, variable accessible to all functions of that file, but not functions of another file.

37) Memory Leak: -

In computer science, a memory leak is a type of resource leak that occurs when a computer program incorrectly manages memory allocations[1] in such a way that memory which is no longer needed is not released. A memory leak may also happen when an object is stored in memory but cannot be accessed by the running code.[2] A memory leak has symptoms similar to a number of other problems and generally can only be diagnosed by a programmer with access to the programs' source code.

The memory leak happen when we allocate memory in head using dynamic memory allocation. After uses this we does not free or release this concept is called memory leak.

38) how would you declare a C variable without defining it?

Ans: - **extern int var;** //without extern keyword variable get declare and define.

39) Print "Hello World" Without using semicolon (;)?

Ans: - **if(printf("Hello World")){}**

40) What is the difference between the const and volatile qualifier in C?

Ans: -

- The const keyword is compiler-enforced and says that the program could not change the value of the object that means it makes the object nonmodifiable type.
e.g,
`const int a = 0;`
- if you will try to modify the value of "a", you will get the compiler error because "a" is qualified with const keyword that prevents to change the value of the integer variable.
- In another side volatile prevent from any compiler optimization and says that the value of the object can be changed by something that is beyond the control of the program and so that compiler will not make any assumption about the object.

e.g,
volatile int a;

- When the compiler sees the above declaration then it avoids to make any assumption regarding the “a” and in every iteration read the value from the address which is assigned to the variable.

capability to recognize and process or forward transmissions to other nodes.

41) Why are global variables always initialized to '0', but not local variables?

Ans: -

- global and static variables are stored in the Data Segment (DS) when initialized and block start by symbol (BSS) when uninitialized.
- These variables have a fixed memory location, and memory is allocated at compile time.
- Thus, global and static variables have '0' as their default values.
- Whereas auto variables are stored on the stack, and they do not have a fixed memory location.
- Memory is allocated to auto variables at runtime, but not at compile time. Hence auto variables have their default value as garbage.

42) where is volatile variables stored in memory layout?

Ans: -

- The storage of a variable is not decided based on if it is volatile or not.
- It can be anywhere as is the case with normal variable. So the variable may be on stack, heap or in the data section of executable, depending on how it gets defined.
- The volatile qualifier just tells the compiler that this variable may change in ways that are not apparent to you, it can be changed at any point of time, for example is hardware registers (variables mapped to h/w registers).
- So compiler disables any optimizations such as caching on this variable.
- Volatile is type qualifier not a storage class specifier.

43) Volatile Memory:

Volatile memory is a computer storage that only maintain its data when power is on.
Eg. RAM is a volatile memory

Non-Volatile Memory:

Nonvolatile memory is a computer storage that maintain its data when the power is not available.