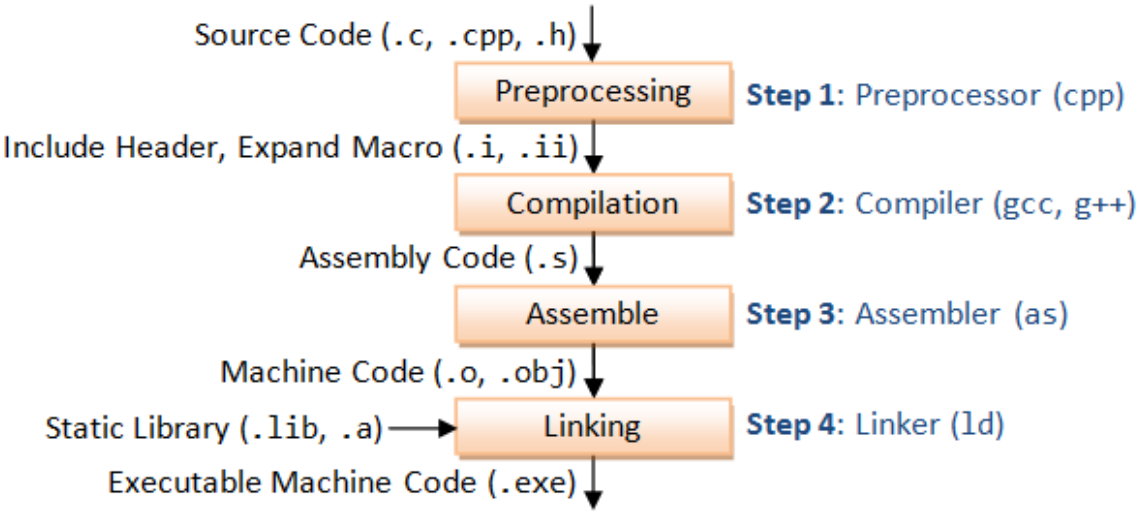
1. Explain about compilation process in C.

The gcc compilation process can be broken down into four steps and each temporary file gets generated after a step gets completed while the executable is generated after the last step.



These steps are:

1. Preprocessing
2. Compilation
3. Assemble
4. Linking

Let's understand the basics of these steps one by one.

1. The preprocessing step

- In this stage all the header files that you have included in your program are actually expanded and included in source code of your program.
- Other than this, all the macros are replaced by their respective values all over the code and all the comments are stripped off.
- The intermediate file that is generated after this stage is the **.i**file.

2. The compilation step

- In this step, the source code is actually compiled by the compiler to produce an assembly code. Assembly code consist of set of instructions that determine what you program wants to do.
- In earlier days the code was written mostly in assembly language but then higher level languages like C, COBOL etc. were developed as programs can be written at much faster pace in these languages as these languages are easy to understand and program.
- So whenever there is a source code written in a higher level language, the compiler converts the source code into the assembly language. The intermediate file produced at this stage is a **.s** file.

3. The assembly step

- In this step, the assembler understands the assembly instructions and converts each of them in to the corresponding machine level code or a bit stream that consists of 0's and 1's. This machine level code is known as object code and this code can be executed by the processor.

- The object code consists of different sections that the processor uses while executing the program. The intermediate file produced after this step is the **.o**file.

4. The linking step

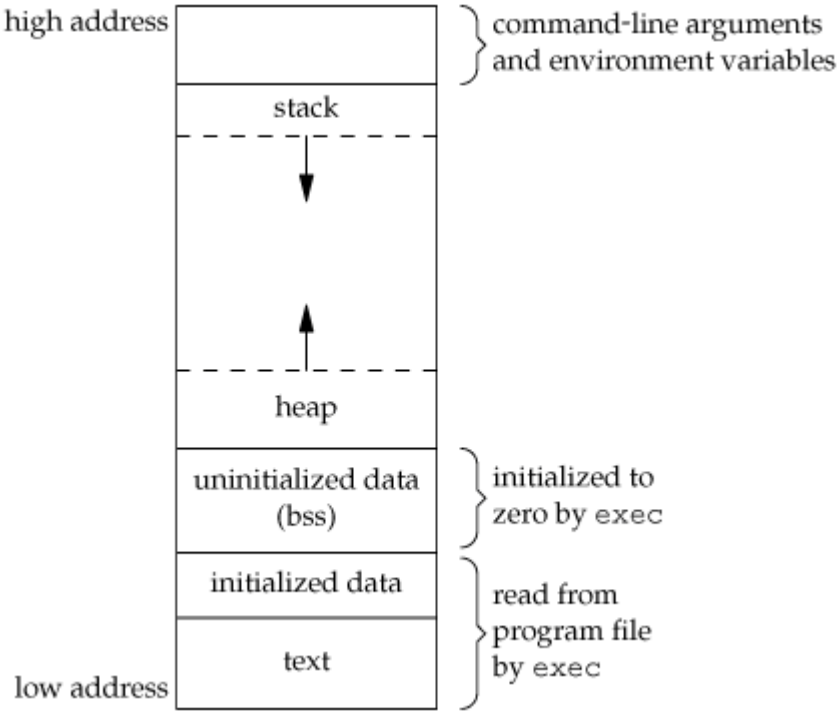
- Once the object file containing the machine code is produced in the step above, the linking step makes sure that all the undefined symbols in code are resolved. An undefined symbol is one for which there is no definition available.
- For example, in a code, there is no definition of printf () function. So in order to make program execute correctly, the definition of this function need to included or at least linked to code. This is what happens in the Linking stage.
- In another example, suppose your source code consists of more than one source files, then the assembly stage produces separate**.o**files corresponding to all the individual **.c** files. Then it's the linking stage where all these object files are linked together. The output after this step is the final executable.

2. Explain about Memory layout in C.

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.
- 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. The Heap area is shared by all shared libraries and dynamically loaded modules in a process.

3. Explain about "Constant" and "Volatile" in C.

const Keyword

In c all variables are by default not constant. Hence, you can modify the value of variable by program. You can convert any variable as a constant variable by using modifier const which is keyword of c language.
Properties of constant variable:
1. You can assign the value to the constant variables only at the time of declaration.
For example:

```
const int i=10;
float const f=0.0f;
unsigned const long double ld=3.14L;
```

2. Uninitialized constant variable is not cause of any compilation error. But you cannot assign any value after the declaration.
For example:

```
const int i;
```

If you have declared the uninitialized variable globally then default initial value will be zero in case of integral data type and null in case of non-integral data type. If you have declared the uninitialized const variable locally then default initial value will be garbage.
3. Constant variables executes faster than not constant variables.

volatile Keyword

volatile is a qualifier that is applied to a variable when it is declared. It tells the compiler that the value of the variable may change at any time-without any action being taken by the code the compiler finds nearby. The implications of this are quite serious. However, before we examine them, let's take a look at the syntax.
Syntax
To declare a variable volatile, include the keyword volatile before or after the data type in the variable definition. For instance both of these declarations will declare "x" to be a volatile integer:

```
volatile int x;
int volatile x;
```

Now, it turns out that pointers to volatile variables are very common. Both of these declarations declare "x" to be a pointer to a volatile integer:

```
volatile int * x;
int volatile * f;
```

Volatile pointers to non-volatile variables are very rare, but the syntax is as below:

```
int * volatile x;
```

A volatile pointer to a volatile variable syntax is:

```
int volatile * volatile x;
```

If we apply volatile to a struct or union, the entire contents of the struct/union are volatile. If we don't want this behavior, we can apply the volatile qualifier to the individual members of the struct/union.

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

A. Memory-mapped peripheral registers
B. Global variables modified by an interrupt service routine

C. Global variables within a multi-threaded application

Peripheral registers

Embedded systems contain real hardware, usually with sophisticated peripherals. These peripherals contain registers whose values may change asynchronously to the program flow. As a very simple example, consider an 8-bit status register at address 0x1234. It is required that you poll the status register until it becomes non-zero. The nave and incorrect implementation is as follows:

```
UINT1 * ptr = (UINT1 *) 0x1234;
// Wait for register to become non-zero.
while (*ptr == 0);
// Do something else.
```

This will almost certainly fail as soon as you turn the optimizer on, since the compiler will generate assembly language that looks something like this:

```
Mov ptr, #0x1234
loop mov    a, @ptr
loop bz
```

The rationale of the optimizer is quite simple: having already read the variable's value into the accumulator (on the second line), there is no need to reread it, since the value will always be the same. Thus, in the third line, we end up with an infinite loop. To force the compiler to do what we want, we modify the declaration to:

```
UINT1 volatile * ptr = (UINT1 volatile *) 0x1234;
```

The assembly language now looks like this:

```
mov ptr, #0x1234
loop mov a, @ptr
bz    loop
```

The desired behavior is achieved.

Subtler problems tend to arise with registers that have special properties. For instance, a lot of peripherals contain registers that are cleared simply by reading them. Extra (or fewer) reads than we are intending can cause quite unexpected results in these cases.

Interrupt service routines

Interrupt service routines often set variables that are tested in main line code. For example, a serial port interrupt may test each received character to see if it is an ETX character (presumably signifying the end of a message). If the character is an ETX, the ISR might set a global flag. An incorrect implementation of this might be:

```
int etx_rcvd = FALSE;
void main()
        {
          ...
      while (!ext_rcvd)
        {
                // Wait
               }
          ...
        }
interrupt void rx_isr(void)
        {
```

```
                ...
if (ETX == rx_char)
        {
         etx_rcvd = TRUE;
        }
          ...
}
```

With optimization turned off, this code might work. However, any half decent optimizer will "break" the code. The problem is that the compiler has no idea that etx_rcvd can be changed within an ISR. As far as the compiler is concerned, the expression !ext_rcvd is always true, and, therefore, we can never exit the while loop. Consequently, all the code after the while loop may simply be removed by the optimizer. The solution is to declare the variable etx_rcvd to be volatile. Then all of your problems (well, some of them anyway) will disappear.

Multi-threaded applications

Despite the presence of queues, pipes, and other scheduler-aware communications mechanisms in real-time operating systems, it is still fairly common for two tasks to exchange information via a shared memory location (that is, a global). When weadd a pre-emptive scheduler to our code, our compiler still has no idea what a context switch is or when one might occur. Thus, another task modifying a shared global is conceptually identical to the problem of interrupt service routines discussed previously. So all shared global variables should be declared volatile.

For example:

```
int cntr;
void task1(void)
{
   cntr = 0;
   while (cntr == 0)
   {
      sleep(1);
   }
   ...
}
void task2(void)
{
   ...
   cntr++;
   sleep(10);
   ...
}
```

This code will likely fail once the compiler's optimizer is enabled. Declaring cntr to be volatile is the proper way to solve the problem.

Final thoughts

Some compilers allow us to implicitly declare all variables as volatile. Resist this temptation, since it is essentially a substitute for thought. It also leads to potentially less efficient code.

Also, resist the temptation to blame the optimizer or turn it off. Modern optimizers are so good.

4. What is Difference between “Structure” and “Union” in C?

Difference between Structure and Union:

S.No	Structure	Union
1	The keyword **struct** is used to define a structure	The keyword union is used to define a union.
2	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. The smaller members may end with unused slack bytes.	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.
3	Each member within a structure is assigned unique storage area of location.	Memory allocated is shared by individual members of union.
4	The address of each member will be in ascending order This indicates that memory for each member will start at different offset values.	The address is same for all the members of a union.This indicates that every member begins at the same offset value.
5	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.
6	Individual member can be accessed at a time	Only one member can be accessed at a time.
7	Several members of a structure can initialize at once.	Only the first member of a union can be initialized.

5. Explain about "Storage Classes" in C.

Storage Classes

Every C variable has a storage class and a scope. The storage class determines the part of memory where storage is allocated for an object and how long the storage allocation continues to exist. It also determines the scope which specifies the part of the program over which a variable name is visible, i.e. the variable is accessible by name.

There are four storage classes in C

1. Automatic (auto)
2. Register (register)
3. External (extern) and
4. Static (static)

Automatic storage class

- Keyword for automatic variable is auto
- Variables declared inside the function body are automatic by default. These variable are also known as local variables as they are local to the function and doesn't have meaning outside that function
- Automatic variables are declared at the start of a block. Memory is allocated automatically upon entry to a block and freed automatically upon exit from the block. The scope of automatic variables is local to the block in which they are declared, including any blocks nested within that block. For these reasons, they are also called local variables. No block outside the defining block may have direct access to

automatic variables, i.e. by name. Of course, they may be accessed indirectly by other blocks and/or functions using pointers.

- Automatic variables may be specified upon declaration to be of storage class auto. However, it is not required; by default, storage class within a block is auto. Automatic variables declared with initializers are initialized each time the block in which they are declared is entered.

External storage class

- Keyword for automatic variable is extern
- External variable can be accessed by any function. They are also known as global variables. Variables declared outside every function are external variables.
- In case of large program, containing more than one file, if the global variable is declared in file 1 and that variable is used in file 2 then, compiler will show error. To solve this problem, keyword extern is used in file 2 to indicate that, the variable specified is global variable and declared in another file.
    - Example to demonstrate working of external variable

```
#include
void Check();
int a=5;
/* a is global variable because it is outside every function */
int main()
{
   a+=4;
   Check();
   return 0;
}
void Check(){
  ++a;
/*  ----- Variable a is not declared in this function but, works in any function as they are global
variable -------  */
  printf("a=%d\n",a);
}
```

Output

```
a=10
```

Register Storage Class

- Keyword to declare register variable is register
- Example of register variable

```
register int a;
```

- Register variables are similar to automatic variable and exists inside that particular function only.
- If the compiler encounters register variable, it tries to store variable in microprocessor's register rather than memory. Value stored in register are much faster than that of memory.
- In case of larger program, variables that are used in loops and function parameters are declared register variables.Since, there are limited number of register in processor and if it couldn't store the variable in register, it will automatically store it in memory.

Static Storage Class

- The value of static variable persists until the end of the program.
- A variable can be declared static using keyword: static.

For example:

static int i; //Here, i is a static variable.

- Example to demonstrate the static variable

```
#include <stdio.h>
void Check();
int main()
{
  Check();
  Check();
  Check();
}
void Check()
{
   static int c=0;
   printf("%d\t",c);
   c+=5;
}
```

Output

0 5 10

- During first function call, it will display 0. Then, during second function call, variable c will not be initialized to 0 again, as it is static variable. So, 5 is displayed in second function call and 10 in third call.
- If variable c had been automatic variable, the output would have been:

0 0 0

6. Write a Program to "Swap of two numbers without using third Variable" in four possible ways.

Program to Swap of two numbers:

```
#include<stdio.h>
int main()
{
        int a=5,b=10;
//process one
        a=b+a;
        b=a-b;
        a=a-b;
        printf("a= %d b= %d",a,b);
//process two
        a=5;
        b=10;
        a=a+b-(b=a);
        printf("\na= %d b= %d",a,b);
//process three
        a=5;
        b=10;
        a=a^b;
```

```
        b=a^b;
        a=b^a;
        printf("\na= %d b= %d",a,b);
//process four
        a=5;
        b=10;
        a=b-~a-1;
        b=a+~b+1;
        a=a+~b+1;
        printf("\na= %d b= %d",a,b);
}
```

7. What is "#pragma" statement?

#pragma:

- '#pragma' is for compiler directives that are machine-specific or operating-system-specific, i.e. it tells the compiler to do something, set some option, take some action, override some default, etc. that may or may not apply to all machines and operating systems.
- This is a preprocessor directive that can be used to turn on or off certain features. It is of two type's #pragma startup, #pragma exit and #pragma warn.
  - #pragma startup allows us to specify functions called upon program startup.
  - #pragma exit allows us to specify functions called upon program exit.
  - #pragma warn tells the computer to suppress any warning or not.

8. What are "Inline functions" in C?

Inline Functions

- Inline function expansion replaces a function call with the function body. With automatic inline function expansion, programs can be constructed with many small functions to handle complexity and then rely on the compilation to eliminate most of the function calls. Therefore, inline expansion serves a tool for satisfying two conflicting goals:
  - Minimizing the complexity of the program development.
  - Minimizing the function call overhead of program execution.
- A simple inline expansion procedure is presented which uses profile information to address three critical issues:
  - Code expansion,
  - Stack expansion
  - Unavailable function bodies
- Inline functions provide the speed of a macro, together with the clarity and type-safety of a function.
- With an inline function is called, the compiler does not insert code to push arguments on a stack and transfer to the code for the function, but rather inserts code to perform the entire computation of the function right at the point of call.
- The overhead of calling the function is eliminated.

- However, the code that is inserted (at the place where the function call occurs) must perform exactly like an ordinary function call. In particular, the arguments are evaluated first.
- An example:

```
inline int max ( int x , int y )
{
        returnx > y ? x : y ;
}
. . .
printf( ”%i \n” , max (20 , 10) ) ;
```

When max is ”called”, the compiler may simply substitutethis code.

```
temp1 = 20;
temp2 = 10;
printf( ”%i \n” temp1 > temp2 ? temp1 : temp2 ) ;
```

Many compilers can optimize this so that it doesn’t requirethe temporaries.

- Inline Advantages

Inline substitution can pay off in several ways. First, it can eliminate the overhead in doing a function call. When a function is called, the following steps are usually taken:

- Argument values are copied to the stack or special registers.
- A return address is created and stored on the stack or to a register.
- The program branches to the function.
- A stack frame is set up for the local variables of the function.
- After the function finishes, the stack frame is torn down.
- The return address is retrieved.
- A branch is made to the return address.

- Summary

Inline substitution is a general optimization that can be controlled to some extent using the C99 inline keyword. Inlining a function produces the same results as a normal call to the function, but may run faster and may permit more optimization than a normal call. Static inline functions have no special considerations, but extern inline functions require that the programmer pick one module to contain a real callable version of the function and follow some restrictions about accessing statics.

9. Explain about “Call by Value” or “Pass by Value” and “Call by Reference” or “Pass by Reference”.

In c we can pass the parameters in a function in two different ways.

(a) Pass by value: In this approach we pass copy of actual variables in function as a parameter. Hence any modification on parameters inside the function will no reflect in the actual variable.

For example:

```
#include<stdio.h>
void main()
{
        int a=5,b=10;
        swap(a,b);
        printf("%d %d",a,b);
}
```

```
void swap(int a,int b)
   {
        int temp;
        temp =a;
        a=b;
        b=temp;
}
Output: 5 10
```

(b)Pass by reference: In this approach we pass memory address actual variables in function as a parameter. Hence any modification on parameters inside the function will reflect in the actual variable.

For example:

```
#include<stdio.h>
void main()
{
        int a=5,b=10;
        swap(&a,&b);
        printf("%d %d",a,b);
}
void swap(int *a,int *b)
   {
        int *temp;
        *temp =*a;
        *a=*b;
        *b=*temp;
}
Output: 10 5
```

10. What are Macros?

Macros

- Preprocessor macros add powerful features and flexibility to C. But they have a downside:
  - Macros have no concept of scope; they are valid from the point of definition to the end of the source. They cut a swath across .h files, nested code, etc. When #include'ing tens of thousands of lines of macro definitions, it becomes problematical to avoid inadvertent macro expansions.
  - Macros are unknown to the debugger. Trying to debug a program with symbolic data is undermined by the debugger only knowing about macro expansions, not the macros themselves.
  - Macros make it impossible to tokenize source code, as an earlier macro change can arbitrarily redo tokens.
  - The purely textual basis of macros leads to arbitrary and inconsistent usage, making code using macros error prone. (Some attempt to resolve this was introduced with templates in C++.)
  - Macros are still used to make up for deficits in the language's expressive capability, such as for "wrappers" around header files.
  - Here's an enumeration of the common uses for macros, and the corresponding feature in D:

- Examples
  - Defining literal constants:

```
#define VALUE 5
```

  - Creating a list of values or flags:

```
int flags:
#define FLAG_X  0x1
#define FLAG_Y  0x2
#define FLAG_Z  0x4
...
flags |= FLAG_X;
```

  - Setting function calling conventions:

```
#ifndef _CRTAPI1
#define _CRTAPI1 __cdecl
#endif
#ifndef _CRTAPI2
#define _CRTAPI2 __cdecl
#endif

int _CRTAPI2 func();
```

  - Simple generic programming:

Selecting which function to use based on text substitution:

```
#ifdef UNICODE
int getValueW(wchar_t *p);
#define getValue getValueW
#else
int getValueA(char *p);
#define getValue getValueA
#endif
```

11. Explain about "Macros and Macro functions".

Difference between Macros and Functions

Sr.No	Macros	Functions

1.	A macro replaces the original text in the source code for each macro call,
	Ex:
	#define square(x) x*x
		Function code can be written only at one place, we can call the function regardless of the number of times.
		Ex:
		int square(int x)
		{
		return x*x;
		}

2.	Macro makes the code lengthy and the compilation time increases.	The use of functions makes the code smaller.
3.	Generally Macros do not extend beyond one line.	Function can be of any number of lines

4.	In macros no arguments passing and no returning values so, the time is saved. Hence the execution of the program becomes fast.	In functions passing the arguments and returning a value takes some time. Hence the execution of the program becomes slow.
5.	Macros are fast but occupy more memory due to duplicity of code.	Functions are slow but take less memory.
6.	Macros just replaces the text without any type checking, so one macro can use with any data type.	Functions perform type checking, so separate functions have to write for every data type.
7.	Macros do not have addresses.	Functions have addresses and we can call using function pointer.
8.	The macros are useful where small code appears many times.	Functions are useful where large code appears many times.

Conclusion:

Hence whether to use a macro or a function, it depends upon our requirement, the memory available and the nature of the task which we are going to perform.

12. Difference between "typedef" and "macro".
    1. The typedef is limited to giving symbolic names to types only where as #define can be used to define alias for values as well, like you can define 1 as ONE etc.
    2. The typedef interpretation is performed by the compiler where as #define statements are processed by the pre-processor.
    3. #define will just copy-paste the definition values at the point of use, while typedef is actual definition of a new type.
    4. typedef follows the scope rule which mean if a new type is defined in a scope(inside a function), then the new type name will only be visible till the scope is there.
13. Difference between macro's and constant.

    The difference between Macro and const Declaration:

    Let's start with: what is/are the difference/s between the following two statements and which of these two is efficient and meaningful?

    ```
    #define SIZE 50
    int const size = 50;
    ```

    Macros

    1. The statement "#define SIZE 50" is a preprocessor directive with SIZE defined as a MACRO. The advantage of declaring a MACRO in a prots for gram is to replace the MACRO name with the Value given to the MACRO in the #define statement wherever MACRO occurs in the program.
    2. No memory is allocated to the MACROS. Program is faster in Execution because of no trade-offs due to allocation of memory!
    3. Their Primary use in the program is where constant values viz. characters, integers, floating point is to be used. For example: as an array subscripts

    ```
    #define MAX 50    /* Declaring a MACRO MAX */
    /* MAX is a MACRO which is replaced with value 50 wherever occurs in the program*/
    int roll_no[MAX];
    ```

    4. A MACRO Statement does not terminate with a semicolon " ; "

Constants

The statement "int const size = 50;" declares and defines size to be a constant integer with the value 50. The const keyword causes the identifier size to be allocated in the read-only memory. This means that the value of the identifier can not be changed by the executing program.

MACROS are efficient than the const statements as they are not given any memory, being more Readable and Faster in execution!

14. Difference between enum, typedef and macro?

enum:

The enumerated data type is designed for variables that contain only a limited set of values. These values are referenced by name (tag). The compiler assigns each tag an integer value internally. Consider an application, for example, in which we want a variable to hold the days of the week. We could use the const declaration to create values for the week_days, as follows:

```
typedef int week_day; /* define the type for week_days */
const int SUNDAY = 0;
const int MONDAY = 1;
const int TUESDAY = 2;
const int WEDNESDAY = 3;
const int THURSDAY = 4;
const int FRIDAY = 5;
const int SATURDAY = 6;
/* now to use it */
week_day today = TUESDAY;
```

This method is cumbersome. A better method is to use the enum type:

```
enum week_day {SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY,
FRIDAY, SATURDAY};
/* now use it */
enum week_day today = TUESDAY;
```

The general form of an enum statement is:

```
enum enum-name { tag-1, tag-2, . . .} variable-name
```

Like structures, the enum-name or the variable-name may be omitted. The tags may be any valid C identifier; however, they are usually all uppercase.

C implements the enum type as compatible with integer. So in C, it is perfectly legal to say:

```
today = 5; /* 5 is not a week_day */
```

although some compilers will issue a warning when they see this line. In C++, enum is a separate type and is not compatible with integer.

Differences between Enumeration and Macro are:

- Enumeration is a type of integer.
- Macros can be of any type. Macros can even be any code block containing statements, loops, function calls etc.
- Syntax/Example of Enumeration:

```
enum
```

```
{
 element1, /*Default 0*/
 element2,
 element3 = 5,
};
```

- #define WORD unsigned short
- Macros are expanded by the preprocessor before compilation takes place. Compiler will refer error messages in expanded macro to the line where the macro has been called.

```
Code:
Line:1          :#define set_userid(x)\
Line:2:                 current_user = x \
Line:3:                 next_user = x + 1;
Line:4:                 int main (int argc, char *argv[])
Line:5:                 {
Line:6:                 int user = 10;
Line:7:                 set_userid(user);
Line:7:                 }
main.c  (7):error C2146: syntax error : missing ';' before identifier 'next_user'
```

Line 2 has the original compilation error

Difference between Enum and typedef statement

Typedef statement allows user to define an identifier that would represent anexiting data type. The user-defined data type identifier can be used further todeclare variables. It has the following syntax;

Code:

```
typedef datatype identifier;
```

where datatype refers to exiting data type and identifier is the new name given to this datatype.

For example

Code:

```
typedef int nos;
```

nos here symbolizes int type and now it can be used later to declare variables like

Code:

```
nos num1,num2,num3;
```

enum statement is used to declare variables that can have one of the valuesenclosed within braces known as enumeration constant. After declaring this, we can declare variables to be of this 'new' type. It has the following syntax

Code:

```
enum identifier {value1, value2, value3……,valuen);
```

where value1, value2 etc are values of the identifier. For example

Code:

```
enum day { Monday, Tuesday, Wednesday, Thursday, Friday,Saturday}
```

We can define later

Code:

```
enum day working_day;
working_day=Monday;
```

The compiler automatically assigns integer digits beginning with 0 to allenumeration constants.

15. What is function pointer?

Function Pointer

Just like variables have address functions also have address. This means that we can make use of a pointer to represent the address of a function as well. If we use a pointer to represent the address of an integer we declare it as int* p. Similarly if it is a character pointer we declare it as char* p. But if it is a function pointer how can we declare it.

Suppose we have a function as

```
void add(int x, int y)
{
printf("Value = %d", x + y);
}
```

if we want to declare a pointer to represent this function we can declare it as

```
void (*p)(int x, int y);
```

The above declaration means that p is a function pointer that can point to any function whose return type is void and takes two integer arguments. The general syntax for a function pointer declaration will look like,

<return type> (<Pointer Variable>) (<data type> [variable name], <data type> [variablename]);

Variable names in the argument list are optional. It is only the data type that matters. Having seen how we can declare a function pointer the next step is to assign the address of a function to this pointer. But how do we get the address of a function? Just like an array whenever we refer a function by its name we actually refer to the address of the function. This means that we can assign the function address as,

```
p = add;
```

Note that we are not specifying the function parenthesis for add. If we put a function parenthesis it means that we are calling a function. Now how can we use this function pointer? Wherever we can call add we can use this function pointer interchangeably i.e. we can say,

```
printf("Value = %d", (*p)(10, 20));
```

Which will make a call to add and print the result. Since it is a pointer it can be made to point to different functions at different places. i.e. if we have another method sub declared as

```
void sub(int x., int y)
{
printf("Value = %d", x - y);
}
```

Then we can say,

```
p = sub;
```

and call it as

```
printf("Value = %d", (*p)(20, 10));
```

This time it will make a call to sub and print the result as 10.

Advantages

a. It gives direct control over memory and thus we can play around with memory by all possible means. For example we can refer to any part of the hardware like keyboard, video memory, printer, etc directly.
b. As working with pointers is like working with memory it will provide enhanced performance

c. Pass by reference is possible only through the usage of pointers
d. Useful while returning multiple values from a function
e. Allocation and freeing of memory can be done wherever required and need not be done in advance

Limitations

a. If memory allocations are not freed properly it can cause memory leakages
b. If not used properly can make the program difficult to understand

16. Bit Addressing mode

a. Set a particular bit and clear a particular bit?

C has direct support for bitwise operations that can be used for bit manipulation. In the following examples, n is the index of the bit to be manipulated within the variable bit_fld, which is an unsigned char being used as a bit field. Bit indexing begins at 0, not 1. Bit 0 is the least significant bit.

Set a bit:

```
bit_fld |= (1 << n)
```

Clear a bit:

```
bit_fld &= ~ (1 << n)
```

b. Swap a nibble in a byte?

Swap a Nibble:

```
((x & 0x0F) <<4 | (x & 0xF0)>>4);
```

c. Toggle a particular bit?

Toggle a bit

```
bit_fld ^= (1 << n)
```

Test a bit

```
bit_fld & (1 << n)
```

d. Swap odd and even no?

Swap odd and even digits of Number:

```
#include<stdio.h>
int swap_even_odd_bits(int num)
    {
      int even_bits = num&0xAAAAAAAA;
      even_bits = even_bits>>1;

      int odd_bits = num&0x55555555;
      odd_bits = odd_bits<<1;
      return even_bits|odd_bits;
    }

int main()
    {
      unsigned int x = 23; // 00010111
      printf("%u ", swap_even_odd_bits(x));
```

```
    return 0;
}
```

e. Find given no is odd or even using bitwise operators?

To check odd or even using bitwise operator

```
#include <stdio.h>
int main()
{
  int n;
  printf("Enter an integer\n");
  scanf("%d", &n);
  if (n & 1 == 1)
    printf("Odd\n");
  else
    printf("Even\n");
  return 0;
}
```

To check odd or even without using bitwise or modulus operator

```
#include <stdio.h>
int main()
{
  int n;
 printf("Enter an integer\n");
  scanf("%d", &n);

  if ((n/2)*2 == n)
    printf("Even\n");
  else
    printf("Odd\n");

  return 0;
}
```

f. Masking in bits?

- A mask defines which bits you want to keep, and which bits you want to clear. Masking is the act of applying a mask to a value. This is accomplished by doing:
  - Bitwise **AND**ing in order to extract a subset of the bits in the value
  - Bitwise **OR**ing in order to set a subset of the bits in the value
  - Bitwise **XOR**ing in order to toggle a subset of the bits in the value
- Below is an example of extracting a subset of the bits in the value:

Mask: 00001111b

Value: 01010101b

- Applying the mask to the value means that we want to clear the first (higher) 4 bits, and keep the last (lower) 4 bits. Thus we have extracted the lower 4 bits. The result is:

  Mask: 00001111b
  Value: 01010101b
  Result: 00000101b

- Masking is implemented using AND, so in C we get:

```
uint8_t stuff(...)
        {
                uint8_t mask = 0x0f;   // 00001111b
                uint8_t value = 0x55;  // 01010101b
                return mask & value;
        }
```

17. What is void pointer?

- Void pointer or generic pointer is a special type of pointer that can be pointed at objects of any data type. A void pointer is declared like a normal pointer, using the void keyword as the pointer's type.
- Pointers defined using specific data type cannot hold the address of the some other type of variable i.e., it is incorrect in C++ to assign the address of an integer variable to a pointer of type float.

  Example:

```
float *f; //pointer of type float
int i;  //integer variable
f = &i; //compilation error
```

- The above problem can be solved by general purpose pointer called void pointer.
- Void pointer can be declared as follows:

```
void *v // defines a pointer of type void
```

- The pointer defined in this manner do not have any type associated with them and can hold the address of any type of variable.

  Example:

```
void *v;
int *i;
int ivar;
char chvar;
float fvar;
v = &ivar; // valid
v = &chvar; //valid
v = &fvar; // valid
i = &ivar; //valid
i = &chvar; //invalid
i = &fvar; //invalid
```

18. What is null pointer?

- NULL Pointer is a pointer which is pointing to nothing.
- NULL pointer points the base address of segment.

- In case, if you don't have address to be assigned to pointer then you can simply use NULL
- Pointer which is initialized with NULL value is considered as NULL pointer.
- NULL is macro constant defined in following header files -
  stdio.h
  alloc.h
  mem.h
  stddef.h
  stdlib.h
- Defining NULL Value
  #define NULL 0
- We have noticed that 0 value is used by macro pre-processor. It is re-commanded that you should not use NULL to assign 0 value to a variable.

```
#include<stdio.h>
int main()
{
    int num = NULL;
    printf("Value of Number ", ++num);
    return 0;
}
```

- Above program will never print 1, so keep in mind that NULL should be used only when you are dealing with pointer.
- Below are some of the variable representations of NULL pointer.

```
float *ptr  = (float *)0;
char  *ptr  = (char *)0;
double *ptr = (double *)0;
char *ptr   = '\0';
int *ptr    = NULL;
```

you can see that we have assigned the NULL to only pointers.

- Example of NULL Pointer

```
#include <stdio.h>
int main()
{
  int  *ptr = NULL;
  printf("The value of ptr is %u",ptr);

  return 0;
}
```

Output :

The value of ptr is 0

- <u>Uses:</u>

The null pointer is used in three ways:

1. To stop indirection in a recursive data structure
2. As an error value
3. As a sentinel value

19. What is null character?

- The null character '\0', whose value is 0, is used to mark the end of a string, which is defined by the C standard as "a contiguous sequence of characters terminated by and including the first null character." For example, the strlen() function, which returns the length of a string, works by scanning through the sequence of characters until it finds the terminating null character.

20. Extracting bits of a variable using structure?

Bit fields in C

- There are times when the member variables of a structure represent some flags that store either 0 or 1.

Here is an example :

```
struct info
{
   int isMemoryFreed;
   int isObjectAllocated;
}
```

- If you observe, though a value of 0 or 1 would be stored in these variables but the memory used would be complete 8 bytes.
- To reduce memory consumption when it is known that only some bits would be used for a variable, the concept of bit fields can be used.
- Bit fields allow efficient packaging of data in the memory. Here is how bit fields are defined :

```
struct info
{
   int isMemoryFreed : 1;
   int isObjectAllocated : 1;
}
```

- The above declaration tells the compiler that only 1 bit each from the two variables would be used. After seeing this, the compiler reduces the memory size of the structure.

Here is an example that illustrates this :

```
#include <stdio.h>

struct example1
{
   int isMemoryAllocated;
   int isObjectAllocated;
};

struct example2
{
   int isMemoryAllocated : 1;
```

```
    int isObjectAllocated : 1;
};

int main(void)
{
    printf("\n sizeof example1 is [%u], sizeof example2 is [%u]\n", sizeof(struct example1),
     sizeof(struct example2));
    return 0;
}
```

Here is the output :

sizeof example1 is [8], sizeof example2 is [4]

- Also, if after declaring the bit field width (1 in case of above example), if you try to access other bits then compiler would not allow you to do the same.

Here is an example :

```
#include <stdio.h>
struct example2
{
   int isMemoryAllocated : 1;
   int isObjectAllocated : 1;
};

int main(void)
{
   struct example2 obj;

   obj.isMemoryAllocated = 2;

  return 0;
}
```

- So, by setting the value to '2', we try to access more than 1 bits. Here is what compiler complains:

```
$ gcc -Wall bitf.c -o bitf
bitf.c: In function 'main':
bitf.c:14:5: warning: overflow in implicit constant conversion [-Woverflow]
```

- So we see that compiler effectively treats the variables size as 1 bit only.

21. What is structure padding? Is structure padding is advantage (or) disadvantage? If so Why and How?

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. Because 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.
- Below C program is compiled and executed in 32 bit compiler. Please check memory allocated for structure1 and structure2 in below program.

  Example program for structure padding in C:

```
#include <stdio.h>
#include <string.h>
/*  Below structure1 and structure2 are same.
They differ only in member's alignment */
struct structure1
   {
      int id1;
      int id2;
      char name;
      char c;
      float percentage;
   };
struct structure2
   {
      int id1;
      char name;
      int id2;
```

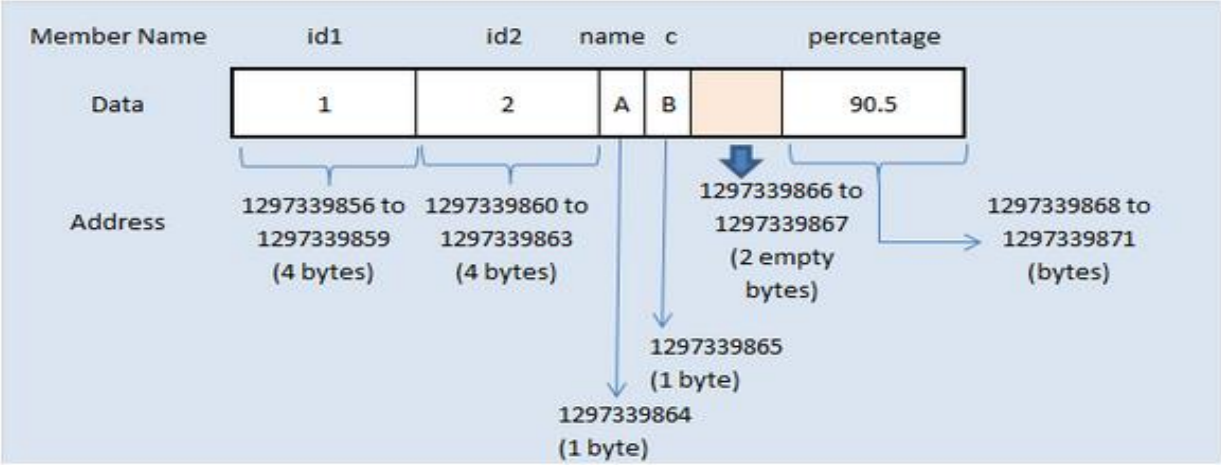
```
        char c;
        float percentage;
    };
int main()
{
    struct structure1 a;
    struct structure2 b;
    printf("size of structure1 in bytes : %d\n", sizeof(a));
    printf ( "\n   Address of id1       = %u", &a.id1 );
    printf ( "\n   Address of id2       = %u", &a.id2 );
    printf ( "\n   Address of name      = %u", &a.name );
    printf ( "\n   Address of c        = %u", &a.c );
    printf ( "\n   Address of percentage = %u",&a.percentage );
    printf("   \n\nsize of structure2 in bytes : %d\n", sizeof(b));
    printf ( "\n   Address of id1       = %u", &b.id1 );
    printf ( "\n   Address of name      = %u", &b.name );
    printf ( "\n   Address of id2       = %u", &b.id2 );
    printf ( "\n   Address of c        = %u", &b.c );
    printf ( "\n   Address of percentage = %u",&b.percentage );
    getchar();
    return 0;
}
```

Output:

```
size of structure1 in bytes     : 16
Address of id1                  = 1297339856
Address of id2                  = 1297339860
Address of name                 = 1297339864
Address of c            = 1297339865
Address of percentage           = 1297339868
sizeof structure2 in bytes      : 20
Address of id1                  = 1297339824
Address of name                 = 1297339828
Address of id2                  = 1297339832
Address of c            = 1297339836
Address of percentage           = 1297339840
```

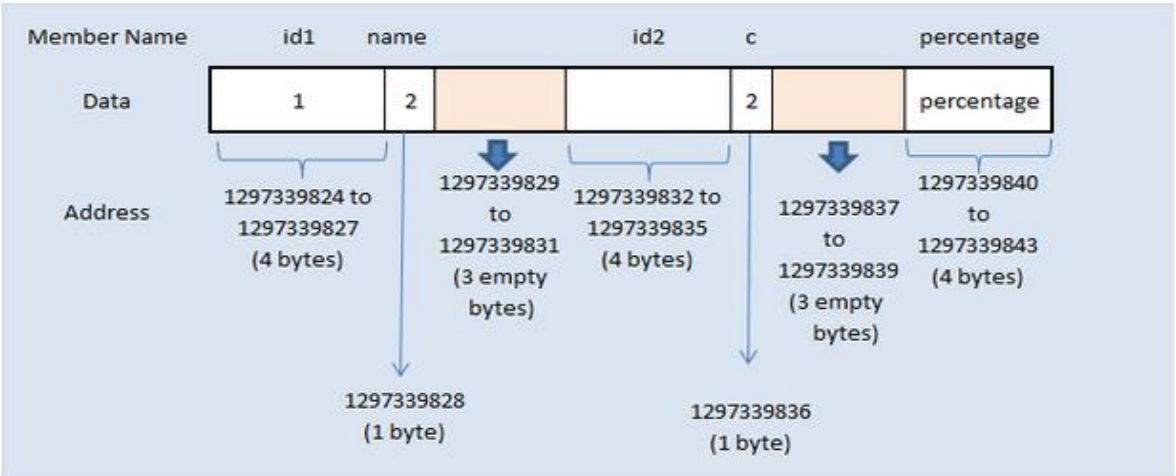
Structure padding analysis for above C program:

Memory allocation for structure1:



- In above program, memory for structure1 is allocated sequentially for first 4 members.
- Whereas, memory for 5th member "percentage" is not allocated immediate next to the end of member "c".
- There are only 2 bytes remaining in the package of 4 bytes after memory allocated to member "c".
- Range of this 4 byte package is from 1297339864 to 1297339867.
- Addresses 1297339864 and 1297339865 are used for members "name and c". Addresses 1297339866 and 1297339867 only is available in this package.
- But, member "percentage" is datatype of float and requires 4 bytes. It can't be stored in the same memory package as it requires 4 bytes. Only 2 bytes are free in that package.
- So, next 4 byte of memory package is chosen to store percentage data which is from 1297339868 to 1297339871.
- Because of this, memory 1297339866 and 1297339867 are not used by the program and those 2 bytes are left empty.
- So, size of structure1 is 16 bytes which is 2 bytes extra than what we think. Because, 2 bytes are left empty.

Memory allocation for structure2:



- Memory for structure2 is also allocated as same as above concept. Please note that structure1 and structure2 are same. But, they differ only in the order of the members declared inside the structure.
- 4 bytes of memory is allocated for 1st structure member “id1” which occupies whole 4 byte of memory package.
- Then, 2nd structure member “name” occupies only 1 byte of memory in next 4 byte package and remaining 3 bytes are left empty. Because, 3rd structure member “id2” of datatype integer requires whole 4 byte of memory in the package. But, this is not possible as only 3 bytes available in the package.
- So, next whole 4 byte package is used for structure member “id2”.
- Again, 4th structure member “c” occupies only 1 byte of memory in next 4 byte package and remaining 3 bytes are left empty.
- Because, 5th structure member “percentage” of datatype float requires whole 4 byte of memory in the package.
- But, this is also not possible as only 3 bytes available in the package. So, next whole 4 byte package is used for structure member “percentage”.
- So, size of structure2 is 20 bytes which is 6 bytes extra than what we think. Because, 6 bytes are left empty.
- Structure padding is adding extra bits at the end of the structure, so that the structure completes the word boundary.

How to avoid structure padding in C:

#pragma:

- #pragma issues special commands to the compiler, using a standardized method.
  #pragma pack (1) directive can be used for arranging memory for structure members very next to the end of other structure members.
- Some compilers such as VC++ supports this feature. But, some compilers such as Turbo C/C++ does not support this feature.
- Please check the below program where there will be no addresses (bytes) left empty because of structure padding.

Example program to avoid structure padding in C:

```
#include <stdio.h>
#include <string.h>
/*  Below structure1 and structure2 are same.
They differ only in member's alignment */
#pragma pack (1)
struct structure1
{
    int id1;
    int id2;
    char name;
    char c;
    float percentage;
};
struct structure2
{
```

```
    int id1;
    char name;
    int id2;
    char c;
    float percentage;
};

int main()
{
    struct structure1 a;
    struct structure2 b;
    printf("size of structure1 in bytes : %d\n", sizeof(a));
    printf ( "\n   Address of id1       = %u", &a.id1 );
    printf ( "\n   Address of id2       = %u", &a.id2 );
    printf ( "\n   Address of name      = %u", &a.name );
    printf ( "\n   Address of c         = %u", &a.c );
    printf ( "\n   Address of percentage = %u", &a.percentage );
    printf(" \n\n size of structure2 in bytes : %d\n",sizeof(b));
    printf ( "\n   Address of id1       = %u", &b.id1 );
    printf ( "\n   Address of name      = %u", &b.name );
    printf ( "\n   Address of id2       = %u", &b.id2 );
    printf ( "\n   Address of c         = %u", &b.c );
    printf ( "\n   Address of percentage = %u", &b.percentage );
    getchar ();
    return 0;
}
```

Output:

```
size of structure1 in bytes     : 14
Address of id1                  = 3438103088
Address of id2                  = 3438103092
Address of name                 = 3438103096
Address of c            = 3438103097
Address of percentage           = 3438103098
size of structure2 in bytes     : 14
Address of id1                  = 3438103072
Address of name                 = 3438103076
Address of id2                  = 3438103077
Address of c            = 3438103081
Address of percentage           = 3438103082
```

- The #pragma preprocessor directive allows each compiler to implement compiler-specific features that can be turned on and off with the #pragma statement. For instance, your compiler might support a

feature called loop optimization. This feature can be invoked as a command-line option or as a #pragma directive.

- To implement this option using the #pragma directive, you would put the following line into your code:
  #pragma loop -opt(on)
- Conversely, you can turn off loop optimization by inserting the following line into your code:
  #pragma loop -opt(off)

  we can use better way of #pragma pre processor
- In case of structure memory is padding while declaring of structure, to overcome this problem better way....we write our code as follows
  #pragma pack(push)
  #pragma pack(1)

  place this before structure we can avoid padding in structure.

22. What is memory leakage in 'C'? How to reduce it?

Memory leak occurs when programmers create a memory in heap and forget to delete it.

Memory leaks are particularly serious issues for programs like daemons and servers which by definition never terminate.

/* Function with memory leak */

```
#include <stdlib.h>
 void f()
{
  int *ptr = (int *) malloc(sizeof(int));

  /* Do some work */

  return; /* Return without freeing ptr*/
}
```

To avoid memory leaks, memory allocated on heap should always be freed when no longer needed.

/* Function without memory leak */

```
#include <stdlib.h>;
 void f()
{
  int *ptr = (int *) malloc(sizeof(int));

  /* Do some work */

  free(ptr);
  return;
}
```

23. What is dynamic memory allocation?

C Programming Dynamic Memory Allocation

- The exact size of array is unknown until the compile time,i.e., time when a compiler compiles code written in a programming language into a executable form. The size of array you have declared initially can be sometimes insufficient and sometimes more than required. Dynamic memory allocation allows a program to obtain more memory space, while running or to release space when no space is required.

- Although, C language inherently does not has any technique to allocated memory dynamically, there are 4 library functions under "stdlib.h" for dynamic memory allocation.

Function	Use of Function
malloc()	Allocates requested size of bytes and returns a pointer first byte of allocated space
calloc()	Allocates space for an array elements, initializes to zero and then returns a pointer to memory
realloc()	Change the size of previously allocated space
free()	De-allocate the previously allocated space

malloc()

- The name malloc stands for "memory allocation". The function malloc() reserves a block of memory of specified size and return a pointer of type void which can be casted into pointer of any form.
- Syntax of malloc()

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

Here, ptr is pointer of cast-type. The malloc() function returns a pointer to an area of memory with size of byte size. If the space is insufficient, allocation fails and returns NULL pointer.

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

- This statement will allocate either 200 or 400 according to size of int 2 or 4 bytes respectively and the pointer points to the address of first byte of memory.

calloc()

- The name calloc stands for "contiguous allocation". The only difference between malloc() and calloc() is that, malloc() allocates single block of memory whereas calloc() allocates multiple blocks of memory each of same size and sets all bytes to zero.
- Syntax of calloc()

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

- This statement will allocate contiguous space in memory for an array of n elements. For example:

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

- This statement allocates contiguous space in memory for an array of 25 elements each of size of float, i.e, 4 bytes.

free()

- Dynamically allocated memory with either calloc() or malloc() does not get return on its own. The programmer must use free() explicitly to release space.
- syntax of free()

free(ptr);

- This statement cause the space in memory pointer by ptr to be deallocated.

24. Explain about "malloc (), calloc (), alloc () and free" with examples?

Dynamic memory allocation functions in C:

C language offers 4 dynamic memory allocation functions. They are,

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

S.no	Function	Syntax
1	malloc ()	malloc (number *sizeof(int));
2	calloc ()	calloc (number, sizeof(int));
3	realloc ()	realloc (pointer_name, number * sizeof(int));
4	free ()	free (pointer_name);

1. malloc() function in C:

- malloc () function is used to allocate space in memory during the execution of the program.
- malloc () does not initialize the memory allocated during execution. It carries garbage value.
- malloc () function returns null pointer if it couldn't able to allocate requested amount of memory.

- Example program for malloc() function in C:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int main()
{
   char *mem_allocation;
   /* memory is allocated dynamically */
   mem_allocation = malloc( 20 * sizeof(char) );
   if( mem_allocation== NULL )
   {
     printf("Couldn't able to allocate requested memory\n");
   }
   else
   {
     strcpy( mem_allocation,"mallac()");
   }
   printf("Dynamically allocated memory content : " \
```

```
            "%s\n", mem_allocation );
        free(mem_allocation);
    }
```

Output:
Dynamically allocated memory content : mallac()

2. Calloc () function in C:

- Calloc () function is also like malloc () function. But calloc () initializes the allocated memory to zero. But, malloc () doesn't.
- Example program for calloc() function in C:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int main()
{
    char *mem_allocation;
    /* memory is allocated dynamically */
    mem_allocation = calloc( 20, sizeof(char) );
    if( mem_allocation== NULL )
    {
      printf("Couldn't able to allocate requested memory\n");
    }
    else
    {
       strcpy( mem_allocation,"callac()");
    }
       printf("Dynamically allocated memory content   : " \
            "%s\n", mem_allocation );
       free(mem_allocation);
}
```

Output:
Dynamically allocated memory content : callac()

3. realloc() function in C:

- realloc () function modifies the allocated memory size by malloc () and calloc () functions to new size.

If enough space doesn't exist in memory of current block to extend, new block is allocated for the full size of reallocation, then copies the existing data to new block and then frees the old block.

4. free() function in C:

- free () function frees the allocated memory by malloc (), calloc (), realloc () functions and returns the memory to the system.

- Example program for realloc() and free() functions in C:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int main()
{
   char *mem_allocation;
   /* memory is allocated dynamically */
   mem_allocation = malloc( 20 * sizeof(char) );
   if( mem_allocation == NULL )
   {
      printf("Couldn't able to allocate requested memory\n");
   }
   else
   {
     strcpy( mem_allocation,"fresh2refresh.com");
   }
   printf("Dynamically allocated memory content  : " \
         "%s\n", mem_allocation );
   mem_allocation=realloc(mem_allocation,100*sizeof(char));
   if( mem_allocation == NULL )
   {
      printf("Couldn't able to allocate requested memory\n");
   }
   else
   {
      strcpy( mem_allocation,"space is extended upto " \
                        "100 characters");
   }
   printf("Resized memory : %s\n", mem_allocation );
   free(mem_allocation);
}
```

Output:
Dynamically allocated memory content : realloc() and free()
Resized memory : space is extended upto 100 characters

25. Declare the following

a. Declare a variable

```
int a; // An integer
```

b. Declare an array

```
int a[10]; // An array of 10 integers
```

c. Declare a pointer variable

```
int *a; // A pointer to an integer
```

d. Declare array of pointers

```
int *a[10]; // An array of 10 pointers to integers
```

e. Declare pointer to arrays

```
int (*a)[10]; // A pointer to an array of 10 integers
```

f. Declare array and functions

C Programming Arrays and Functions

In C programming, a single array element or an entire array can be passed to a function. Also, both one-dimensional and multi-dimensional array can be passed to function as argument.

Passing One-dimensional Array in Function

C program to pass a single element of an array to function

```
#include <stdio.h>
void display(int a)
  {
  printf("%d",a);
  }
int main()
{
  int c[]={2,3,4};
  display(c[2]);  //Passing array element c[2] only.
  return 0;
}
```

Output

4

Single element of an array can be passed in similar manner as passing variable to a function.

Passing entire one-dimensional array to a function

While passing arrays to the argument, the name of the array is passed as an argument(,i.e, starting address of memory area is passed as argument).

Write a C program to pass an array containing age of person to a function. This function should find average age and display the average age in main function.

```
#include <stdio.h>
float average(float a[]);
int main()
{
    float avg, c[]={23.4, 55, 22.6, 3, 40.5, 18};
    avg=average(c);   /* Only name of array is passed as argument. */
    printf("Average age=%.2f",avg);
    return 0;
  }
float average(float a[])
{
    int i;
```

```
    float avg, sum=0.0;
    for(i=0;i<6;++i)
    {
      sum+=a[i];
    }
    avg =(sum/6);
    return avg;
}
```

Output

Average age=27.08

Passing Multi-dimensional Arrays to Function

To pass two-dimensional array to a function as an argument, starting address of memory area reserved is passed as in one dimensional array

Example to pass two-dimensional arrays to function

```
#include
void Function(int c[2][2]);
int main()
{
  int c[2][2],i,j;
  printf("Enter 4 numbers:\n");
  for(i=0;i<2;++i)
    for(j=0;j<2;++j)
{
          scanf("%d",&c[i][j]);
}
  Function(c);   /* passing multi-dimensional array to function */
  return 0;
}
void Function(int c[2][2])
{
/* Instead to above line, void Function(int c[][2]){ is also valid */
  int i,j;
  printf("Displaying:\n");
  for(i=0;i<2;++i)
    for(j=0;j<2;++j)
      printf("%d\n",c[i][j]);
}
```

Output

```
Enter 4 numbers:
2
3
4
```

```
5
Displaying:
2
3
4
5
```

g. Declare function with arrays

Same as f answer

h. Declare function with pointer arguments

```
#include <stdio.h>
int func(int *a, int *b);  // function declaration
intmain()
{
  int a = 4, b = 7;
func(&a,&b);          // function calling
}

Int func(int *a, int *b)   // function defination
{
   - -
 - -
    - -
}
```

i. Declare array of structures

- C Structure is collection of different datatypes (variables) which are grouped together. Whereas, array of structures is nothing but collection of structures. This is also called as structure array in C.
- Example program for array of structures in C:
- This program is used to store and access “id, name and percentage” for 3 students. Structure array is used in this program to store and display records for many students. You can store “n” number of students record by declaring structure variable as ‘struct student record[n]“, where n can be 1000 or 5000 etc.

```
#include <stdio.h>
#include <string.h>

struct student
{
    int id;
    char name[30];
    float percentage;
};
```

```
int main()
{
   int i;
   struct student record[2];

   // 1st student's record
   record[0].id=1;
   strcpy(record[0].name, "Raju");
   record[0].percentage = 86.5;

   // 2nd student's record
   record[1].id=2;
   strcpy(record[1].name, "Surendren");
   record[1].percentage = 90.5;

   // 3rd student's record
   record[2].id=3;
   strcpy(record[2].name, "Thiyagu");
   record[2].percentage = 81.5;

   for(i=0; i<3; i++)
   {
      printf("     Records of STUDENT : %d \n", i+1);
      printf(" Id is: %d \n", record[i].id);
      printf(" Name is: %s \n", record[i].name);
      printf(" Percentage is: %f\n\n",record[i].percentage);
   }
   return 0;
}
```

Output:

```
Records of STUDENT : 1
Id is: 1
Name is: Raju
Percentage is: 86.500000
Records of STUDENT : 2
Id is: 2
Name is: Surendren
Percentage is: 90.500000
Records of STUDENT : 3
Id is: 3
Name is: Thiyagu
Percentage is: 81.500000
```

j. Declare pointer to structures

Pointers to Structures

You can define pointers to structures in very similar way as you define pointer to any other variable as follows:

```
struct Books *struct_pointer;
```

Now, you can store the address of a structure variable in the above defined pointer variable. To find the address of a structure variable, place the & operator before the structure's name as follows:

```
struct_pointer = &Book1;
```

To access the members of a structure using a pointer to that structure, you must use the -> operator as follows:

```
struct_pointer->title;
```

Let us re-write above example using structure pointer, hope this will be easy for you to understand the concept:

```
#include <stdio.h>
#include <string.h>

struct Books
{
  char  title[50];
  char  author[50];
  char  subject[100];
  int   book_id;
};
/* function declaration */
void printBook( struct Books *book );
int main( )
{
  struct Books Book1;       /* Declare Book1 of type Book */
  struct Books Book2;       /* Declare Book2 of type Book */
  /* book 1 specification */
  strcpy( Book1.title, "C Programming");
  strcpy( Book1.author, "Nuha Ali");
  strcpy( Book1.subject, "C Programming Tutorial");
  Book1.book_id = 6495407;
  /* book 2 specification */
  strcpy( Book2.title, "Telecom Billing");
  strcpy( Book2.author, "Zara Ali");
  strcpy( Book2.subject, "Telecom Billing Tutorial");
  Book2.book_id = 6495700;
  /* print Book1 info by passing address of Book1 */
  printBook( &Book1 );
  /* print Book2 info by passing address of Book2 */
  printBook( &Book2 );
```

```
   return 0;
}
void printBook( struct Books *book )
{
   printf( "Book title : %s\n", book->title);
   printf( "Book author : %s\n", book->author);
   printf( "Book subject : %s\n", book->subject);
   printf( "Book book_id : %d\n", book->book_id);
}
```

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

```
Book title : C Programming
Book author : Nuha Ali
Book subject : C Programming Tutorial
Book book_id : 6495407
Book title : Telecom Billing
Book author : Zara Ali
Book subject : Telecom Billing Tutorial
Book book_id : 6495700
```

k. Declare pointer structures

Pointer within Structure in C Programming:

- Structure may contain the Pointer variable as member.
- Pointers are used to store the address of memory location.
- They can be de-referenced by '*' operator.
- Example :

```
struct Sample
{
    int  *ptr;  //Stores address of integer Variable
    char *name; //Stores address of Character String
}s1;
```

s1 is structure variable which is used to access the "structure members".

```
s1.ptr  = &num;
s1.name = "Pritesh"
```

- Here num is any variable but it's address is stored in the Structure member ptr (Pointer to Integer)
- Similarly Starting address of the String "Pritesh" is stored in structure variable name(Pointer to Character array).
- Whenever we need to print the content of variable num , we are dereferancing the pointer variable num.

```
printf("Content of Num : %d ",*s1.ptr);
printf("Name : %s",s1.name);
```

- Live Example : Pointer Within Structure

```
#include<stdio.h>
struct Student
{
    int  *ptr;  //Stores address of integer Variable
```

```
    char *name; //Stores address of Character String
}s1;
int main()
{
    int roll = 20;
    s1.ptr  = &roll;
    s1.name  = "Pritesh";

    printf("\nRoll Number of Student : %d",*s1.ptr);
    printf("\nName of Student       : %s",s1.name);
    return(0);
}
```

Output :

Roll Number of Student : 20

Name of Student : Pritesh

- Some Important Observations :

printf("\nRoll Number of Student : %d",*s1.ptr);

- We have stored the address of variable 'roll' in a pointer member of structure thus we can access value of pointer member directly using de-reference operator.

printf("\nName of Student : %s",s1.name);

- Similarly we have stored the base address of string to pointer variable 'name'. In order to de-reference a string we never use de-reference operator.

I. Passing a structure to a function

C Programming Structure and Function

- In C, structure can be passed to functions by two methods:
  - Passing by value (passing actual value as argument)
  - Passing by reference (passing address of an argument)

Passing structure by value

- A structure variable can be passed to the function as an argument as normal variable. If structure is passed by value, change made in structure variable in function definition does not reflect in original structure variable in calling function.
- Write a C program to create a structure student, containing name and roll. Ask user the name and roll of a student in main function. Pass this structure to a function and display the information in that function.

```
#include <stdio.h>
struct student{
   char name[50];
   int roll;
};
void Display(struct student stu);
/* function prototype should be below to the structure declaration otherwise compiler
shows error */
int main(){
   struct student s1;
   printf("Enter student's name: ");
```

```
    scanf("%s",&s1.name);
    printf("Enter roll number:");
    scanf("%d",&s1.roll);
    Display(s1);   // passing structure variable s1 as argument
    return 0;
}
void Display(struct student stu){
  printf("Output\nName: %s",stu.name);
  printf("\nRoll: %d",stu.roll);
}
```

Output
Enter student's name: Kevin Amla
Enter roll number: 149
Output
Name: Kevin Amla
Roll: 149

Passing structure by reference

- The address location of structure variable is passed to function while passing it by reference. If structure is passed by reference, change made in structure variable in function definition reflects in original structure variable in the calling function.
- Write a C program to add two distances(feet-inch system) entered by user. To solve this program, make a structure. Pass two structure variable (containing distance in feet and inch) to add function by reference and display the result in main function without returning it.

```
#include <stdio.h>
struct distance{
   int feet;
   float inch;
};
void Add(struct distance d1,struct distance d2, struct distance *d3);
int main()
{
   struct distance dist1, dist2, dist3;
   printf("First distance\n");
   printf("Enter feet: ");
   scanf("%d",&dist1.feet);
   printf("Enter inch: ");
   scanf("%f",&dist1.inch);
   printf("Second distance\n");
   printf("Enter feet: ");
   scanf("%d",&dist2.feet);
   printf("Enter inch: ");
   scanf("%f",&dist2.inch);
```

```
    Add(dist1, dist2, &dist3);

/*passing structure variables dist1 and dist2 by value whereas passing structure variable
dist3 by reference */
    printf("\nSum of distances = %d\'-%.1f\"",dist3.feet, dist3.inch);
    return 0;
}
void Add(struct distance d1,struct distance d2, struct distance *d3)
{
/* Adding distances d1 and d2 and storing it in d3 */
    d3->feet=d1.feet+d2.feet;
    d3->inch=d1.inch+d2.inch;
    if (d3->inch>=12) {    /* if inch is greater or equal to 12, converting it to feet. */
       d3->inch-=12;
       ++d3->feet;
    }
}
```

Output

```
First distance
Enter feet: 12
Enter inch: 6.8
Second distance
Enter feet: 5
Enter inch: 7.5
Sum of distances = 18'-2.3"
```

- Explanation

In this program, structure variables dist1 and dist2 are passed by value (because value of dist1 and dist2 does not need to be displayed in main function) and dist3 is passed by reference ,i.e, address of dist3 (&dist3) is passed as an argument. Thus, the structure pointer variable d3 points to the address of dist3. If any change is made in d3 variable, effect of it is seed in dist3 variable in main function.

m. Structure of functions

No Answer

n. Constant pointer

Constant Pointers

A constant pointer is a pointer that cannot change the address its holding. In other words, we can say that once a constant pointer points to a variable then it cannot point to any other variable.

A constant pointer is declared as follows :

<type of pointer> * const <name of pointer>

An example declaration would look like :

```
int * const ptr;
```

Lets take a small code to illustrate these type of pointers :

```
#include<stdio.h>

int main(void)
{
   int var1 = 0, var2 = 0;
   int *const ptr = &var1;
   ptr = &var2;
   printf("%d\n", *ptr);

   return 0;
}
```

o. Pointer to Constant

Pointer to Constant

As evident from the name, a pointer through which one cannot change the value of variable it points is known as a pointer to constant. These type of pointers can change the address they point to but cannot change the value kept at those address.

A pointer to constant is defined as :

```
const <type of pointer>* <name of pointer>
```

An example of definition could be :

```
const int* ptr;
```

Lets take a small code to illustrate a pointer to a constant :

```
#include<stdio.h>

int main(void)
{
   int var1 = 0;
   const int* ptr = &var1;
   *ptr = 1;
   printf("%d\n", *ptr);

   return 0;
}
```

p. Pointer to Constant pointer

No Answer

q. Constant Pointer to a Constant

Constant Pointer to a Constant

If you have understood the above two types then this one is very easy to understand as its a mixture of the above two types of pointers. A constant pointer to constant is a pointer that can neither change the address its pointing to and nor it can change the value kept at that address.

A constant pointer to constant is defined as :

const <type of pointer>* const <name of pointer>

for example :

const int* const ptr;

Lets look at a piece of code to understand this :

```
#include<stdio.h>

int main(void)
{
   int var1 = 0,var2 = 0;
   const int* const ptr = &var1;
   *ptr = 1;
   ptr = &var2;
   printf("%d\n", *ptr);

   return 0;
}
```

r. How to pass the function as an argument to the function

No Answer

26. Linked List

a. Create Singly Linked List

Program to Create Singly Linked List .

```
#include<stdio.h>
#include<conio.h>
#include<alloc.h>
//-----------------------------------------------
struct node
{
  int data;
  struct node *next;
}*start=NULL;
//-----------------------------------------------

void creat()
{
```

```
char ch;
 do
 {
        struct node *new_node,*current;

        new_node=(struct node *)malloc(sizeof(struct node));

        printf("nEnter the data : ");
        scanf("%d",&new_node->data);
        new_node->next=NULL;

        if(start==NULL)
              {
                 start=new_node;
                 current=new_node;
              }
        else
              {
                current->next=new_node;
                current=new_node;
              }

        printf("nDo you want to creat another : ");
        ch=getche();
        }while(ch!='n');
}
//-------------------------------------------------------

void display()
{
    struct node *new_node;
     printf("The Linked List : n");
     new_node=start;
     while(new_node!=NULL)
      {
           printf("%d--->",new_node->data);
      new_node=new_node->next;
      }
     printf("NULL");
}
//-------------------------------------------------
void main()
{
```

```
            create();
            display();
        }
        //-------------------------------------------------
        Output :

        Enter the data : 10
        Do you want to creat another :  y
        Enter the data : 20
        Do you want to creat another : y

        Enter the data : 30
        Do you want to creat another : n

        The Linked List :
            10--->20--->30--->NULL
```

b. Insert a Node in the beginning of the list.

Insert node at Start/First Position in Singly Linked List

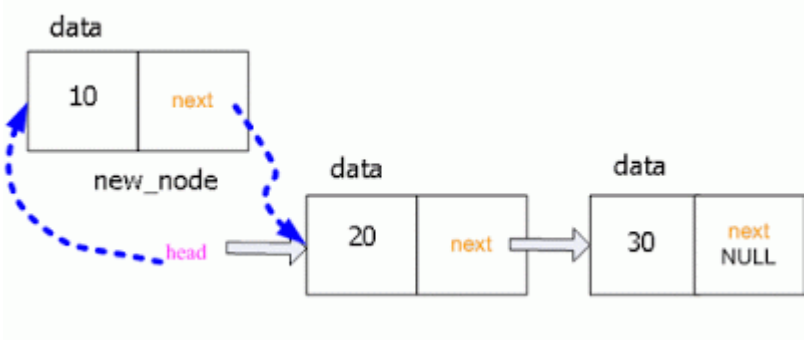
Inserting node at start in the SLL (Steps):

Create New Node
Fill Data into “Data Field“
Make it’s “Pointer” or “Next Field” as NULL
Attach This newly Created node to Start
Make newnode as Starting node

```
void insert_at_beg()
{
struct node *new_node,*current;

new_node=(struct node *)malloc(sizeof(struct node));

 if(new_node == NULL)
   printf("nFailed to Allocate Memory");

 printf("nEnter the data : ");
 scanf("%d",&new_node->data);
 new_node->next=NULL;

  if(start==NULL)
  {
  start=new_node;
```

```
current=new_node;
}
else
{
new_node->next=start;
start=new_node;
}
}
```

Diagram:



Attention :

If starting node is not available then "Start = NULL" then following part is executed

```
if(start==NULL)
    {
  start=new_node;
  current=new_node;
    }
```

If we have previously created First or starting node then "else part" will be executed to insert node at start

```
else
    {
  new_node->next=start;
  start=new_node;
    }
```

c. Insert a Node in the mid of the list.

Insert node at middle position : Singly Linked List

Linked-List : Insert Node at Middle Position in Singly Linked List

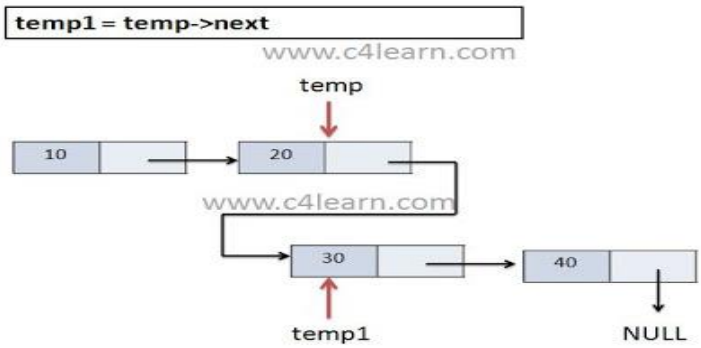
```
void insert_mid()
{
   int pos,i;
   struct node *new_node,*current,*temp,*temp1;

   new_node=(struct node *)malloc(sizeof(struct node));

   printf("nEnter the data : ");
```

```
    scanf("%d",&new_node->data);
    new_node->next=NULL;
    st :
    printf("nEnter the position : ");
    scanf("%d",&pos);
    if(pos>=(length()+1))
      {
      printf("nError : pos > length ");
      goto st;
      }
    if(start==NULL)
      {
      start=new_node;
      current=new_node;
      }
    else
      {
      temp = start;
          for(i=1;i< pos-1;i++)
          {
          temp = temp->next;
          }
      temp1=temp->next;
      temp->next = new_node;
      new_node->next=temp1;
      }
}
```

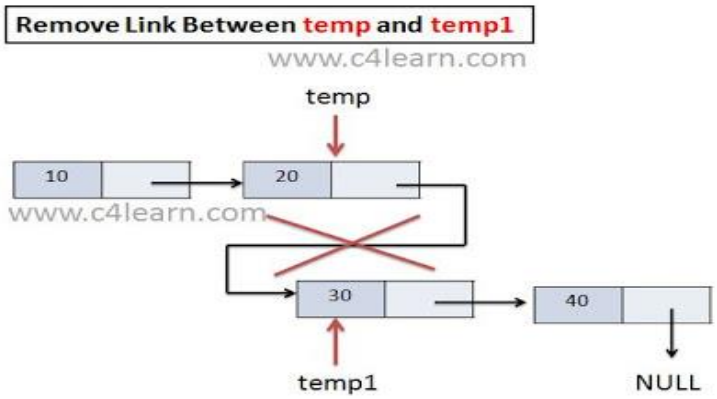
Explanation :

Step 1 : Get Current Position Of “temp” and “temp1” Pointer.

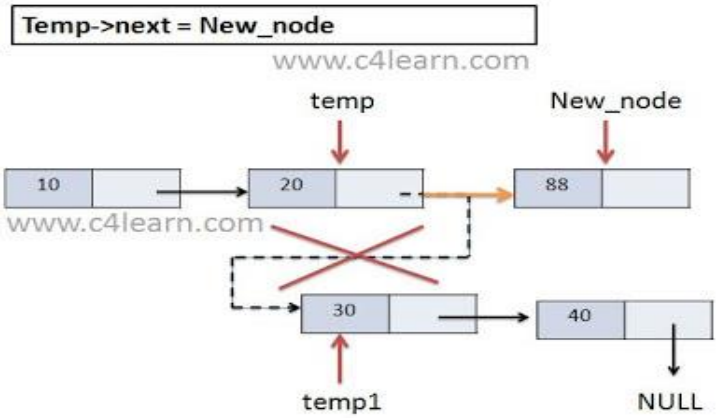
```
temp = start;
          for(i=1;i< pos-1;i++)
 {
 temp = temp->next;
 }
```



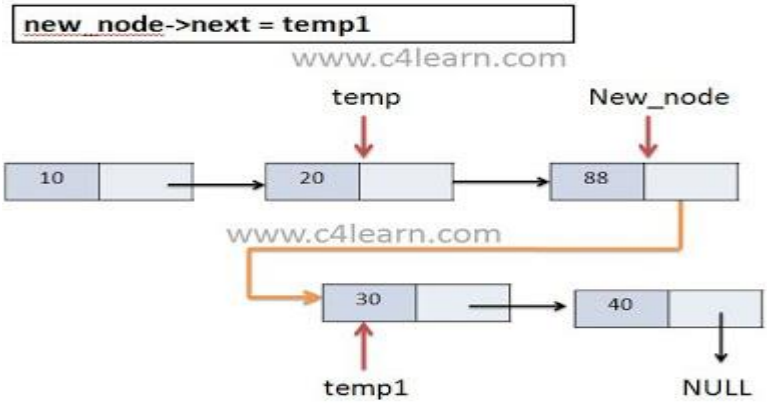
Step 2 :
temp1=temp->next;



Step 3 :
temp->next = new_node;



Step 4 :
new_node->next = temp1



d. Insert a Node anywhere in the list.

No Answer

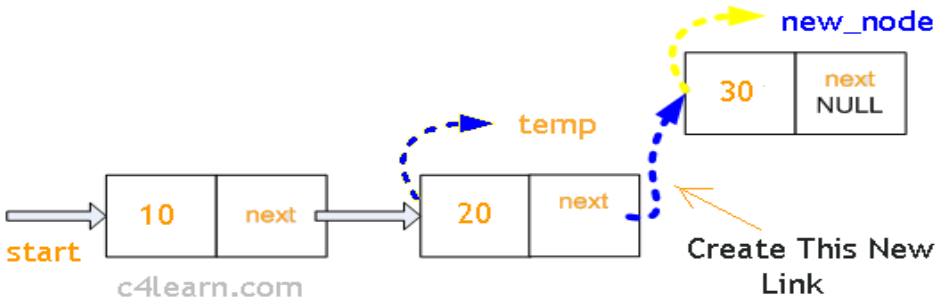
e. Insert a Node in the end of the list.

Insert node at Last Position : Singly Linked List
Insert node at Last / End Position in Singly Linked List
Inserting node at start in the SLL (Steps):

1. Create New Node
2. Fill Data into “Data Field“
3. Make it’s “Pointer” or “Next Field” as NULL
4. Node is to be inserted at Last Position so we need to traverse SLL upto Last Node.
5. Make link between last node and newnode

```
void insert_at_end()
{
struct node *new_node,*current;
new_node=(struct node *)malloc(sizeof(struct node));
if(new_node == NULL)
 printf("nFailed to Allocate Memory");
 printf("nEnter the data : ");
 scanf("%d",&new_node->data);
 new_node->next=NULL;
 if(start==NULL)
 {
  start=new_node;
  current=new_node;
 }
 else
 {
  temp = start;
    while(temp->next!=NULL)
    {
    temp = temp->next;
    }
  temp->next = new_node;
 }
}
```

Diagram :



Attention :

If starting node is not available then “Start = NULL” then following part is executed

i. if(start==NULL)
ii. {
iii. start=new_node;
iv. current=new_node;
v. }

2. If we have previously created First or starting node then “else part” will be executed to insert node at start
3. Traverse Upto Last Node., So that temp can keep track of Last node

```
else
    {
    temp = start;
  while(temp->next!=NULL)
 {
 temp = temp->next;
 }
```

4. Make Link between Newly Created node and Last node (temp)

```
temp->next = new_node;
```

To pass Node Variable to Function Write it as -

```
void insert_at_end(struct node *temp)
```

F.Delete a Node in the beginning of the list.

Delete First Node from Singly Linked List

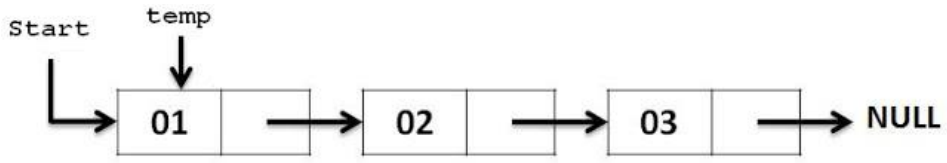
Program :

```
void del_beg()
{
struct node *temp;

temp = start;
start = start->next;

free(temp);
printf("nThe Element deleted Successfully ");
}
```

Attention :

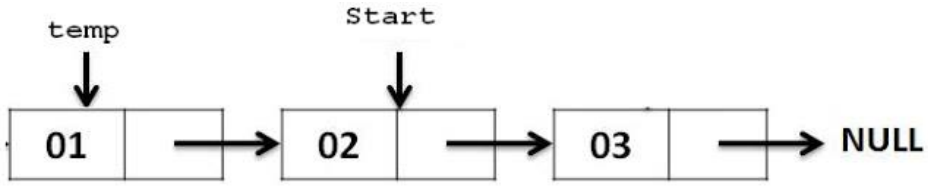
Step 1 : Store Current Start in Another Temporary Pointer

```
temp = start;
```



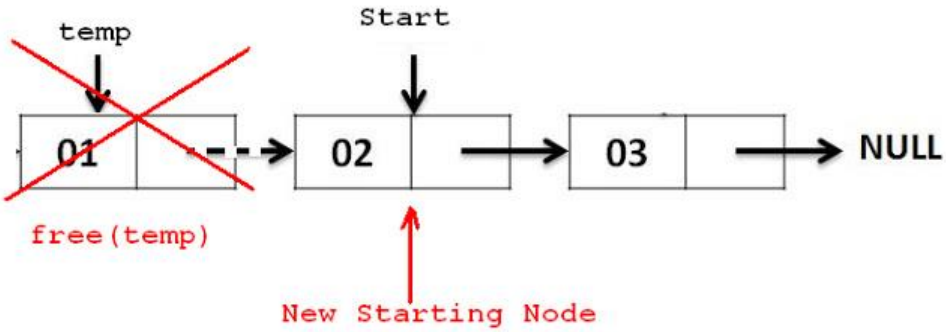
Step 2 : Move Start Pointer One position Ahead

start = start->next;



Step 3 : Delete temp i.e Previous Starting Node as we have Updated Version of Start Pointer

free(temp);



f. Delete a Node in the mid of the list.

g. Delete a Node in the end of the list.

27. Create a Double Linked list for

a. Transverse from node 1 to node N

b. Transverse from node N to node 1

c. Delete Operations

d. Insert Operations

e. Copy a single linked list to another linked list

```
/*
 * C Program to Implement a Doubly Linked List & provide Insertion, Deletion & Display Operations
 */
#include <stdio.h>
#include <stdlib.h>

struct node
{
```

```
   struct node *prev;
   int n;
   struct node *next;
}*h,*temp,*temp1,*temp2,*temp4;

void insert1();
void insert2();
void insert3();
void traversebeg();
void traverseend(int);
void sort();
void search();
void update();
void delete();

int count = 0;

void main()
{
   int ch;

   h = NULL;
   temp = temp1 = NULL;

   printf("\n 1 - Insert at beginning");
   printf("\n 2 - Insert at end");
   printf("\n 3 - Insert at position i");
   printf("\n 4 - Delete at i");
   printf("\n 5 - Display from beginning");
   printf("\n 6 - Display from end");
   printf("\n 7 - Search for element");
   printf("\n 8 - Sort the list");
   printf("\n 9 - Update an element");
   printf("\n 10 - Exit");

   while (1)
   {
      printf("\n Enter choice : ");
      scanf("%d", &ch);
      switch (ch)
      {
      case 1:
         insert1();
```

```
            break;
        case 2:
            insert2();
            break;
        case 3:
            insert3();
            break;
        case 4:
            delete();
            break;
        case 5:
            traversebeg();
            break;
        case 6:
            temp2 = h;
            if (temp2 == NULL)
                printf("\n Error : List empty to display ");
            else
            {
                printf("\n Reverse order of linked list is : ");
                traverseend(temp2->n);
            }
            break;
        case 7:
            search();
            break;
        case 8:
            sort();
            break;
        case 9:
            update();
            break;
        case 10:
            exit(0);
        default:
            printf("\n Wrong choice menu");
        }
    }
}

/* TO create an empty node */
void create()
{
```

```
    int data;

    temp =(struct node *)malloc(1*sizeof(struct node));
    temp->prev = NULL;
    temp->next = NULL;
    printf("\n Enter value to node : ");
    scanf("%d", &data);
    temp->n = data;
    count++;
}

/*  TO insert at beginning */
void insert1()
{
    if (h == NULL)
    {
        create();
        h = temp;
        temp1 = h;
    }
    else
    {
        create();
        temp->next = h;
        h->prev = temp;
        h = temp;
    }
}

/* To insert at end */
void insert2()
{
    if (h == NULL)
    {
        create();
        h = temp;
        temp1 = h;
    }
    else
    {
        create();
        temp1->next = temp;
        temp->prev = temp1;
```

```
      temp1 = temp;
   }
}

/* To insert at any position */
void insert3()
{
   int pos, i = 2;

   printf("\n Enter position to be inserted : ");
   scanf("%d", &pos);
   temp2 = h;

   if ((pos < 1) || (pos >= count + 1))
   {
      printf("\n Position out of range to insert");
      return;
   }
   if ((h == NULL) && (pos != 1))
   {
      printf("\n Empty list cannot insert other than 1st position");
      return;
   }
   if ((h == NULL) && (pos == 1))
   {
      create();
      h = temp;
      temp1 = h;
      return;
   }
   else
   {
      while (i < pos)
      {
         temp2 = temp2->next;
         i++;
      }
      create();
      temp->prev = temp2;
      temp->next = temp2->next;
      temp2->next->prev = temp;
      temp2->next = temp;
   }
```

```
}

/* To delete an element */
void delete()
{
   int i = 1, pos;
   printf("\n Enter position to be deleted : ");
   scanf("%d", &pos);
   temp2 = h;
   if ((pos < 1) || (pos >= count + 1))
   {
      printf("\n Error : Position out of range to delete");
      return;
   }
   if (h == NULL)
   {
      printf("\n Error : Empty list no elements to delete");
      return;
   }
   else
   {
      while (i < pos)
      {
         temp2 = temp2->next;
         i++;
      }
      if (i == 1)
      {
         if (temp2->next == NULL)
         {
            printf("Node deleted from list");
            free(temp2);
            temp2 = h = NULL;
            return;
         }
      }
      if (temp2->next == NULL)
      {
         temp2->prev->next = NULL;
         free(temp2);
         printf("Node deleted from list");
         return;
      }
```

```
      temp2->next->prev = temp2->prev;
      if (i != 1)
         temp2->prev->next = temp2->next;    /* Might not need this statement if i == 1 check */
      if (i == 1)
         h = temp2->next;
      printf("\n Node deleted");
      free(temp2);
   }
   count--;
}

/* Traverse from beginning */
void traversebeg()
{
   temp2 = h;

   if (temp2 == NULL)
   {
      printf("List empty to display \n");
      return;
   }
   printf("\n Linked list elements from begining : ");

   while (temp2->next != NULL)
   {
      printf(" %d ", temp2->n);
      temp2 = temp2->next;
   }
   printf(" %d ", temp2->n);
}

/* To traverse from end recursively */
void traverseend(int i)
{
   if (temp2 != NULL)
   {
      i = temp2->n;
      temp2 = temp2->next;
      traverseend(i);
      printf(" %d ", i);
   }
}
```

```
/* To search for an element in the list */
void search()
{
   int data, count = 0;
   temp2 = h;
   if (temp2 == NULL)
   {
      printf("\n Error : List empty to search for data");
      return;
   }
   printf("\n Enter value to search : ");
   scanf("%d", &data);
   while (temp2 != NULL)
   {
      if (temp2->n == data)
      {
         printf("\n Data found in %d position",count + 1);
         return;
      }
      else
          temp2 = temp2->next;
         count++;
   }
   printf("\n Error : %d not found in list", data);
}

/* To update a node value in the list */
void update()
{
   int data, data1;
   printf("\n Enter node data to be updated : ");
   scanf("%d", &data);
   printf("\n Enter new data : ");
   scanf("%d", &data1);
   temp2 = h;
   if (temp2 == NULL)
   {
      printf("\n Error : List empty no node to update");
      return;
   }
   while (temp2 != NULL)
   {
      if (temp2->n == data)
```

```
            {

               temp2->n = data1;
               traversebeg();
               return;
            }
            else
               temp2 = temp2->next;
         }

         printf("\n Error : %d not found in list to update", data);
      }

      /* To sort the linked list */
      void sort()
      {
         int i, j, x;
         temp2 = h;
         temp4 = h;
         if (temp2 == NULL)
         {
            printf("\n List empty to sort");
            return;
         }
         for (temp2 = h; temp2 != NULL; temp2 = temp2->next)
         {
            for (temp4 = temp2->next; temp4 != NULL; temp4 = temp4->next)
            {
               if (temp2->n > temp4->n)
               {
                  x = temp2->n;
                  temp2->n = temp4->n;
                  temp4->n = x;
               }
            }
         }
         traversebeg();
}
```

28. Difference between Array and Linked List?

S. No.	Array	Linked List
1.	Insertions and deletions are difficult.	Insertions and deletions can be done easily.
2.	It needs movements of elements for insertion and deletion.	It does not need movement of nodes for insertion and deletion.
3.	In it space is wasted.	In it space is not wasted.
4.	It is more expensive.	It is less expensive.
5.	It requires less space as only information is stored.	It requires more space as pointers are also stored along with information.
6.	Its size is fixed.	Its size is not fixed.
7.	It cannot be extended or reduced according to requirements.	It can be extended or reduced according to requirements.
8.	Same amount of time is required to access each element.	Different amount of time is required to access each element.
9.	Elements are stored in consecutive memory locations.	Elements may or may not be stored in consecutive memory locations.
10.	If have to go to a particular element then we can reach there directly.	If we have to go to a particular node then we have to go through all those nodes that come before that node.

29. Difference between' C 'and' Embedded C'?

Difference between' C 'and' Embedded C'

- Though C and embedded C appear different and are used in different contexts, they have more similarities than the differences. Most of the constructs are same; the difference lies in their applications.
- C is used for desktop computers, while embedded C is for microcontroller based applications. Accordingly, C has the luxury to use resources of a desktop PC like memory, OS, etc. While programming on desktop systems, we need not bother about memory. However, embedded C has to use with the limited resources (RAM, ROM, I/Os) on an embedded processor. Thus, program code must fit into the available program memory. If code exceeds the limit, the system is likely to crash.
- Compilers for C (ANSI C) typically generate OS dependant executables. Embedded C requires compilers to create files to be downloaded to the microcontrollers/microprocessors where it needs to run. Embedded compilers give access to all resources which is not provided in compilers for desktop computer applications.
- Embedded systems often have the real-time constraints, which is usually not there with desktop computer applications.
- Embedded systems often do not have a console, which is available in case of desktop applications.

- So, what basically is different while programming with embedded C is the mindset; for embedded applications, we need to optimally use the resources, make the program code efficient, and satisfy real time constraints, if any. All this is done using the basic constructs, syntaxes, and function libraries of 'C'.

30. Difference between Array and Structures?

Difference between Arrays and Structures in C

- Both the arrays and structures are classified as structured data types as they provide a mechanism that enable us to access and manipulate data in a relatively easy manner. But they differ in a number of ways listed in table below:

S.No.	Arrays	Structures
1.	An array is a collection of related data elements of same type.	Structure can have elements of different types
2.	An array is a derived data type	A structure is a programmer-defined data type
3.	Any array behaves like a built-in data types. All we have to do is to declare an array variable and use it.	But in the case of structure, first we have to design and declare a data structure before the variable of that type are declared and used.
4.	An array can't have bit fields.	Structure can contain bit fields.
5.	Array has indexed data type.	Structure has object data type.
6.	Array is a collection of a fixed number of components all ofthe same type: it is a **homogeneous** data structure.	Structure is a collection of a fixed number of components of different types. A struct is typically**heterogeneous.**
7.	Static memory allocation. It uses the subscript to access the array elements.	Dynamic memory allocation. It uses the dot (**.**)operator to access the structure members.
8.	Array elements are referred by subscript.	Structure elements are referred by its unique name.

31. Difference between array and structure?

Same answer of 30 th Question

32. What are call back functions?

Callback Function:

The Callback function is a function that is called through a function pointer. If you pass the pointer (address) of a function as an argument to another, when that pointer is used to call the function it points to it is said that a call back is made.

Why Should You Use Callback Functions?

A callback can be used for notifications. For instance, you need to set a timer in your application. Each time the timer expires, your application must be notified. But, the implementer of the time'rs mechanism doesn't know anything about your application. It only wants a pointer to a function with a given prototype, and in using that pointer it makes a callback, notifying your application about the event that has occurred. Indeed, the SetTimer() WinAPI uses a callback function to notify that the timer has expired (and, in case there is no callback function provided, it posts a message to the application's queue).

Another example from WinAPI functions that use callback mechanism is EnumWindow(), which enumerates all the top-level windows on the screen. EnumWindow() iterates over the top-level windows, calling an application-provided function for each window, passing the handler of the window. If the callee returns a value, the iteration continues; otherwise, it stops. EnumWindows() just doesn't care where the callee is and what it does with the handler it passes over. It is only interested in the return value, because based on that it continues its execution or not.

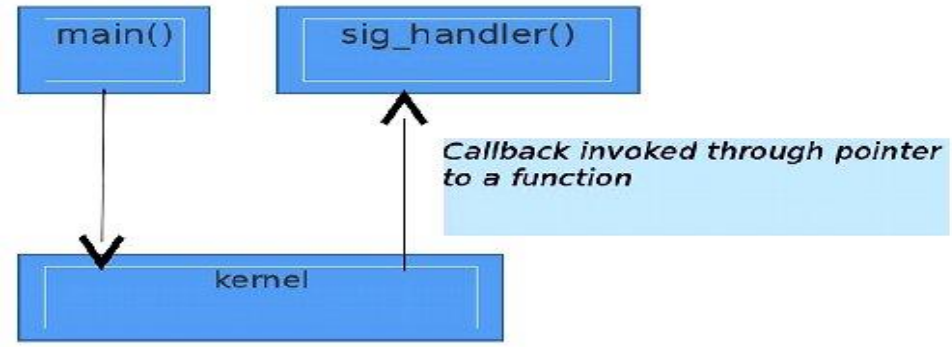
However, callback functions are inherited from C. Thus, in C++, they should be only used for interfacing C code and existing callback interfaces. Except for these situations, you should use virtual methods or functors, not callback functions.

Use of callback functions

One use of callback mechanisms can be seen here:

```
/ * This code catches the alarm signal generated from the kernel
   Asynchronously */
#include <stdio.h>
#include <signal.h>
#include <unistd.h>

struct sigaction act;
/* signal handler definition goes here */
void sig_handler(int signo, siginfo_t *si, void *ucontext)
{
  printf("Got alarm signal %d\n",signo);
  /* do the required stuff here */
}
int main(void)
{
   act.sa_sigaction = sig_handler;
   act.sa_flags = SA_SIGINFO;

   /* register signal handler */
   sigaction(SIGALRM, &act, NULL);
   /* set the alarm for 10 sec */
   alarm(10);
   /* wait for any signal from kernel */
   pause();
   /* after signal handler execution */
   printf("back to main\n");
   return 0;
}
```

Signals are types of interrupts that are generated from the kernel, and are very useful for handling asynchronous events. A signal-handling function is registered with the kernel, and can be invoked asynchronously from the rest of the program when the signal is delivered to the user process. Figure represents this flow.



Kernel callback

Callback functions can also be used to create a library that will be called from an upper-layer program, and in turn, the library will call user-defined code on the occurrence of some event.

33. What is optimization in c?

First things first - don't optimise too early. It's not uncommon to spend time carefully optimising a chunk of code only to find that it wasn't the bottleneck that you thought it was going to be. Or, to put it another way "Before you make it fast, make it work"

Investigate whether there's any option for optimising the algorithm before optimising the code. It'll be easier to find an improvement in performance by optimising a poor algorithm than it is to optimise the code, only then to throw it away when you change the algorithm anyway.

And work out why you need to optimise in the first place. What are you trying to achieve? If you're trying, say, to improve the response time to some event work out if there is an opportunity to change the order of execution to minimise the time critical areas. For example when trying to improve the response to some external interrupt can you do any preparation in the dead time between events?

Once you've decided that you need to optimise the code, which bit do you optimise? Use a profiler. Focus your attention (first) on the areas that are used most often.

So what can you do about those areas?

1. Minimise condition checking. Checking conditions (eg. terminating conditions for loops) is time that isn't being spent on actual processing. Condition checking can be minimised with techniques like loop-unrolling.
2. In some circumstances condition checking can also be eliminated by using function pointers. For example if you are implementing a state machine you may find that implementing the handlers for individual states as small functions (with a uniform prototype) and storing the "next state" by storing the function pointer of the next handler is more efficient than using a large switch statement with the handler code implemented in the individual case statements. YMMV.
3. Minimize function calls. Function calls usually carry a burden of context saving (eg. writing local variables contained in registers to the stack, saving the stack pointer), so if you don't have to make a call this is time saved. One option (if you're optimising for speed and not space) is to make use of inline functions.
4. If function calls are unavoidable minimise the data that is being passed to the functions. For example passing pointers is likely to be more efficient than passing structures.
5. When optimising for speed choose datatypes that are the native size for your platform. For example on a 32bit processor it is likely to be more efficient to manipulate 32bit values than 8 or 16 bit values. (side note - it is worth checking that the compiler is doing what you think it is. I've had situations

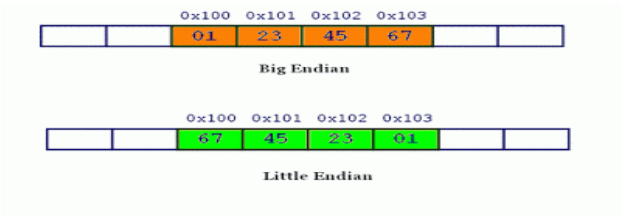
where I've discovered that my compiler insisted on doing 16 bit arithmetic on 8 bit values with all of the to and from conversions to go with them)

6. Find data that can be precalculated, and either calculate during initialisation or (better yet) at compile time. For example when implementing a CRC you can either calculate your CRC values on the fly (using the polynomial directly) which is great for size (but dreadful for performance), or you can generate a table of all of the interim values - which is a much faster implementation, to the detriment of the size.
7. Localise your data. If you're manipulating a blob of data often your processor may be able to speed things up by storing it all in cache. And your compiler may be able to use shorter instructions that are suited to more localised data (eg. instructions that use 8 bit offsets instead of 32 bit)
8. In the same vein, localise your functions. For the same reasons.
9. Work out the assumptions that you can make about the operations that you're performing and find ways of exploiting them. For example, on an 8 bit platform if the only operation that at you're doing on a 32 bit value is an increment you may find that you can do better than the compiler by inlining (or creating a macro) specifically for this purpose, rather than using a normal arithmetic operation.
10. Avoid expensive instructions - division is a prime example.
11. The "register" keyword can be your friend (although hopefully your compiler has a pretty good idea about your register usage). If you're going to use "register" it's likely that you'll have to declare the local variables that you want "register"ed first.
12. Be consistent with your data types. If you are doing arithmetic on a mixture of data types (eg. shorts and ints, doubles and floats) then the compiler is adding implicit type conversions for each mismatch. This is wasted cpu cycles that may not be necessary.
13. Most of the options listed above can be used as part of normal practice without any ill effects. However if you're really trying to eke out the best performance: - Investigate where you can (safely) disable error checking. It's not recommended, but it will save you some space and cycles. - Hand craft portions of your code in assembler. This of course means that your code is no longer portable but where that's not an issue you may find savings here. Be aware though that there is potentially time lost moving data into and out of the registers that you have at your disposal (ie. to satisfy the register usage of your compiler). Also be aware that your compiler should be doing a pretty good job on its own.

34. What is little endian and big endian?

Little and Big Endian Mystery

- Little and big endian are two ways of storing multibyte data-types (int, float, etc). In little endian machines, last byte of binary representation of the multibyte data-type is stored first. On the other hand, in big endian machines, first byte of binary representation of the multibyte data-type is stored first.
- Suppose integer is stored as 4 bytes (For those who are using DOS based compilers such as C++ 3.0 , integer is 2 bytes) then a variable x with value 0x01234567 will be stored as following.



Memory representation of integer ox01234567 inside Big and little endian machines

35. Write a program to find whether the given machine is little endian or big endian?

- There are more number of ways for determining endianness of your machine. Here is one quick way of doing the same.

```
#include <stdio.h>
int main()
{
  unsigned int i = 1;
  char *c = (char*)&i;
  if (*c)
     printf("Little endian");
  else
     printf("Big endian");
  getchar();
  return 0;
}
```

- In the above program, a character pointer c is pointing to an integer i. Since size of character is 1 byte when the character pointer is de-referenced it will contain only first byte of integer. If machine is little endian then *c will be 1 (because last byte is stored first) and if machine is big endian then *c will be 0.

36. Where exactly pointer variables & structures are stored?

Structs (and all value variables) are stored relative to the base of their enclosing scope allocation. Basically this means that when used in a local variable, they are stored on that particular thread's stack and the compiler generates offsets to the current stack frame pointer. When a struct is part of an object variable (class), it is stored as part of that object's data area and referenced relative to the base of the object's memory allocation.

Structs that are part of objects are stored in the heap, but they are not directly garbage collected by the runtime. When the object they are part is garbage collected, the struct effectively disappears from unreachable memory as well.

37. Implement following without using library functions

a. sin(x)

Program to evaluate Sine Series, Cosine Series and Exponential Series?

Sine series : sin x = x - ((x^3)/3!) + ((x^5)/5!) - ……..

Cosine Series : cos x = 1 - ((x^2)/2!) + ((x^4)/4!) -……..

Exponential Series : e^x = 1 + (x/1!) + ((x^2)/2!) + ((x^3)/3!)+…….

- Algorithm
- Step 1:Start
- Step 2:Declare variables i,n,j,ch as integers ;x,t,s,r as float and c='y' as char.
- Step 3:while c='y' repeat steps 4 to 13
- Step 4:Print the menu 1.Sine 2.Cosine 3.Exponential series.
- Step 5:Input and read the choice,ch.
- Step 6:If ch=1 then

6.1)Input and read the limit of the sine series,n.

6.2)Input and read the value of angle(in degree),x.

6.3)Convert the angle in degree to radian,x=((x*3.1415)/180)

6.4)Assign t=r and s=r.
6.5)Initialize i as 2.
6.6)Initialize j as 2.
6.7)while(j<=n) repeat the steps a to d.
a)Find t=(t*(-1)*r*r)/(i*(i+1))
b)Find s=s+t
c)Increment i by 2.
d)Increment j by 1.
6.8)End while.
6.9)Print the sum of the sine series,s.

- Step 7:End if.
- Step 8:Else If the ch=2 then

8.1)Input and read the limit of the cosine series,n.
8.2)Input and read the value of angle(in degree),x.
8.3)Convert the angle in degree to radian,x=((x*3.1415)/180)
8.4)Initialize t as 1,s as 1 and i as 1.
8.5)Initialize j as 2.
8.6)while(j<=n) repeat the steps a to d.
a)Find t=(t*(-1)*r*r)/(i*(i+1))
b)Find s=s+t
c)Increment i by 2.
d)Increment j by 1.
8.7)End while.
8.8)Print the sum of the cosine series,s.

- Step 9:End if.
- Step 10:Else If ch=3 then

10.1)Input and read the limit of the exponential series,n.
10.2)Input and read the value of power of exponential series,x.
10.3)Initialize t as1 and s as 1.
10.4)Initialize i as 1.
10.5)while(i<n) repeat the steps a to c.
a)Find t=(t*x)/i
b)Find s=s+t
c)Increment i by 1.
10.6)End while.
10.7)Print the sum of the exponential series,s.

- Step 11:End if.
- Step 12:Print 'wrong choice'.
- Step 13:End while.
- Step 14:Stop.
- Program

```
#include<stdio.h>
void main()
        {
        int i,n,j,ch;
```

```
float x,t,s,r;
char c='y';
clrscr();
do
{
   printf("\n1.SINE SERIES");
   printf("\n2.COSINE SERIES");
   printf("\n3.EXPONENTIAL SERIES");
   printf("\n ENTER THE CHOICE");
   scanf("%d",&ch);
   switch(ch)
   {
    case 1:
            printf("\nENTER THE LIMIT");
            scanf("%d",&n);
            printf("\nENTER THE VALUE OF x:");
            scanf("%f",&x);
            r=((x*3.1415)/180);
            t=r;
            s=r;
            i=2;
            for(j=2;j<=n;j++)
            {
                t=(t*(-1)*r*r)/(i*(i+1));
                s=s+t;
                i=i+2;
            }
            printf("\nSUM OF THE GIVEN SINE SERIES IS %4f",s);
            break;
    case 2:
            printf("\nENTER THE LIMIT ");
            scanf("%d",&n);
            printf("\nENTER THE VALUE OF x:");
            scanf("%f",&x);
            r=((x*3.14)/180);
            t=1;
            s=1;
            i=1;
            for(j=2;j<=n;j++)
            {
                t=((-1)*t*r*r)/(i*(i+1));
                s=s+t;
                i=i+2;
```

```
                    }
                    printf("\n SUM OF THE COSINE SERIES IS %f",s);
                    break;
                case 3:
                    printf("\nENTER THE LIMIT");
                    scanf("%d",&n);
                    printf("\nENTER THE VALUE OF x:");
                    scanf("%f",&x);
                    t=1;
                    s=1;
                    for(i=1;i<n;i++)
                    {
                        t=(t*x)/i;
                        s=s+t;
                    }
                    printf("\nSUM OF EXPONENTIAL SERIES IS %f",s);
                    break;
                default:
                    printf("\n WRONG CHOICE");
                }
                printf("\n DO U WANT TO CONTINUE Y/N");
                scanf("%c",&c);
            }
            while(c=='y');
            getch();
}
```

Output

1. SINE SERIES

2. COSINE SERIES

3. EXPONENTIAL SERIES

ENTER THE CHOICE **1**

ENTER THE LIMIT **2**

ENTER THE VALUE OF x: **60**

SUM OF THE GIVEN SINE SERIES IS **0.855787**

DO U WANT TO CONTINUE Y/N **N**

b. strcpy( )

Program for String Copy:

Method1

```
#include <stdio.h>
int main()
{
   char s1[100], s2[100], i;
   printf("Enter string s1: ");
   scanf("%s",s1);
   for(i=0; s1[i]!='\0'; ++i)
   {
      s2[i]=s1[i];
   }
   s2[i]='\0';
   printf("String s2: %s",s2);
   return 0;
}
```

Method2

This function copies the string using the pointer, where earlier program copies the string directly.

```
#include<stdio.h>
#include<conio.h>
void stcpy(char *str1, char *str2);
void main()
{
      char *str1, *str2;
      clrscr();
      printf("\n\n\t ENTER A STRING…: ");
      gets(str1);
      stcpy(str1,str2);
      printf("\n\t THE COPIED STRING IS…: ");
      puts(str2);
      getch();
}
void stcpy(char *str1, char *str2)
{
      int i, len = 0;
      while(*(str1+len)!='\0')
      len++;
      for(i=0;i<len;i++)
      *(str2+i) = *(str1+i);
      *(str2+i) = '\0';
}
```

c. strlen( )

Method1

Length of the String using Pointer

```
#include<stdio.h>
int string_ln(char*);
void main()
{
    char str[20];
    int length;
    printf("\nEnter any string : ");
    gets(str);

    length = string_ln(str);
    printf("The length of the given string %s is : %d", str, length);
    getch();
}

int string_ln(char*p) /* p=&str[0] */
{
    int count = 0;
    while (*p != '\0')
    {
        count++;
       p++;
    }
  return count;
}
```

Method2

```
/*PROGRAM TO DETERMINE LENGTH OF STRING USING POINTERS*/
#include<stdio.h>
#include<conio.h>
void main()
{
      char a[10],*p;
      int i=0;
      clrscr();
      printf("Enter any string: ");
      gets(a);
      p=a;
      while(*p!='\0')
      {
                i++;
                p++;
      }
      printf("Length of given string is: %d",i);
```

```
        getch();
    }
```

d. strcmp( )

Method1

C program to compare two strings without using strcmp

Here we create our own function to compare strings.

```
#include <stdio.h>
int compare_strings(char [], char []);
int main()
{
  int flag;
  char a[1000], b[1000];

  printf("Input first string\n");
  gets(a);

  printf("Input second string\n");
  gets(b);

  flag = compare_strings(a, b);

  if (flag == 0)
    printf("Entered strings are equal.\n");
  else
    printf("Entered strings are not equal.\n");

  return 0;
}

int compare_strings(char a[], char b[])
{
  int c = 0;

  while (a[c] == b[c]) {
    if (a[c] == '\0' || b[c] == '\0')
      break;
    c++;
  }

  if (a[c] == '\0' && b[c] == '\0')
    return 0;
  else
    return -1;
}
```

Method2

C program to compare two strings using pointers

In this method we will make our own function to perform string comparison, we will use character pointers in our function to manipulate string.

```
#include<stdio.h>
int compare_string(char*, char*);
int main()
{
   char first[1000], second[1000], result;
   printf("Input first string\n");
   gets(first);
   printf("Input second string\n");
   gets(second);
   result = compare_string(first, second);
   if (result == 0)
     printf("Both strings are same.\n");
   else
     printf("Entered strings are not equal.\n");
   return 0;
}
int compare_string(char *first, char *second)
{
  while (*first == *second)
  {
    if (*first == '\0' || *second == '\0')
      break;
    first++;
    second++;
  }
  if (*first == '\0' && *second == '\0')
    return 0;
  else
    return -1;
}
```

e. strncmp( )

```
int custom_strncmp(const char *s, const char *t, size_t n)
{
   while(n--)
         {
              if(*s != *t)
                   {
                   return *s - *t;
                    }
              else
```

```
                    {
                        ++s;
                        ++t;
                    }
                }
            return 0;
        }
```

f. strncpy( )

```
        char *custom_strncpy(char *s, const char *ct, size_t n)
            {
                char *saver = s;

                while(n--)
                    *saver++ = *ct++;
                *saver = '\0';

                return s;
            }
```

g. strcat( )

```
        void strcat(char s[], char t[])
        {
            int i, j;

            i = j = 0;
            while(s[i] != '\0') /* find the end of s */
                i++;

            while((s[i++] = t[j++]) != '\0'); /*  copy t */
        }
```

h. strncat( )

```
        char *custom_strncat(char *s, const char *t, size_t n, size_t bfsize)
        {
            size_t slen = strlen(s);
            char *pend = s + slen;

            if(slen + n >= bfsize)
                return NULL;

            while(n--)
                *pend++ = *t++;

            return s;
        }
```

38. Print a string without using semicolon (;).

Program:

```
#include <stdio.h>
int main()
 {
    //printf returns the length of string being printed
    if (printf("Hello World\n")) //prints Hello World and returns 11
     {
     //do nothing
     }
     return 0;
 }
```

Output:

Hello World

39. What is conditional compilation?

Conditional compilation in c programming language: Conditional compilation as the name implies code is compiled if certain conditions hold true. Normally we use if keyword for checking some condition so we have to use something different so that compiler can determine whether to compile the code or not. The different thing is #if.

Now consider the following code to quickly understand the scenario:

Conditional compilation example in c language

C programming code 1:

```
#include <stdio.h>
int main()
{
    #define COMPUTER "An amazing device"
            #ifdef COMPUTER
    printf(COMPUTER);
    #endif
            return 0;
}
```

C programming code 2:

```
#include <stdio.h>
#define x 10
main()
{
   #ifdef x
     printf("hello\n");     //this is compiled as  x is defined
   #else
     printf("bye\n");      //this is not compiled
   #endif
            return 0;
}
```

40. Implement recursive function?

Recursive Function

- Recursion is a programming technique that allows the programmer to express operations in terms of themselves. In C, this takes the form of a function that calls itself.
- A useful way to think of recursive functions is to imagine them as a process being performed where one of the instructions is to "repeat the process". This makes it sound very similar to a loop because it repeats the same code, and in some ways it is similar to looping.
- On the other hand, recursion makes it easier to express ideas in which the result of the recursive call is necessary to complete the task. Of course, it must be possible for the "process" to sometimes be completed without the recursive call.
- One simple example is the idea of building a wall that is ten feet high; if I want to build a ten foot high wall, then I will first build a 9 foot high wall, and then add an extra foot of bricks. Conceptually, this is like saying the "build wall" function takes a height and if that height is greater than one, first calls itself to build a lower wall, and then adds one a foot of bricks.
- Example of recursion in C programming

  Write a C program to find sum of first n natural numbers using recursion. Note: Positive integers are known as natural number i.e. 1, 2, 3....n

```
#include <stdio.h>
int sum(int n);
int main(){
   int num,add;
   printf("Enter a positive integer:\n");
   scanf("%d",&num);
   add=sum(num);
   printf("sum=%d",add);
}
int sum(int n){
   if(n==0)
     return n;
   else
     return n+sum(n-1);    /*self call  to function sum() */
}
```

  Output

```
Enter a positive integer:
5
15
```

- In, this simple C program, sum() function is invoked from the same function. If n is not equal to 0 then, the function calls itself passing argument 1 less than the previous argument it was called with. Suppose, n is 5 initially. Then, during next function calls, 4 is passed to function and the value of argument decreases by 1 in each recursive call. When, n becomes equal to 0, the value of n is returned which is the sum numbers from 5 to 1.
- For better visualization of recursion in this example:

```
sum(5)
=5+sum(4)
=5+4+sum(3)
```

=5+4+3+sum(2)
=5+4+3+2+sum(1)
=5+4+3+2+1+sum(0)
=5+4+3+2+1+0
=5+4+3+2+1
=5+4+3+3
=5+4+6
=5+10
=15

- Every recursive function must be provided with a way to end the recursion. In this example when, n is equal to 0, there is no recursive call and recursion ends.
- Advantages and Disadvantages of Recursion
  - Recursion is more elegant and requires few variables which make program clean. Recursion can be used to replace complex nesting code by dividing the problem into same problem of its sub-type.
  - In other hand, it is hard to think the logic of a recursive function. It is also difficult to debug the code containing recursion.

41. Write a Program for given Fibonacci Series?

Program:

```
/* Fibonacci Series c language */
#include<stdio.h>
 int main()
{
  int n, first = 0, second = 1, next, c;
  printf("Enter the number of terms\n");
  scanf("%d",&n);
  printf("First %d terms of Fibonacci series are :-\n",n);
  for ( c = 0 ; c < n ; c++ )
 {
   if ( c <= 1 )
   next = c;
   else
   {
     next = first + second;
     first = second;
     second = next;
    }
  printf("%d\n",next);
   }
 return 0;
}
```

42. Write a Program given number whether it is Palindrome or not?

If a number, which when read in both forward and backward way is same, then such a number is called apalindrome number.

Program:

```
#include <stdio.h>
```

```
int main()
    {
     int n, n1, rev = 0, rem;
     printf("Enter any number: \n");
     scanf("%d", &n);
     n1 = n;
     /* logic */
     while (n > 0)
        {
                rem = n % 10;
                rev = rev * 10 + rem;
                n = n / 10;
         }
    if (n1 == rev)
                {
                        printf("Given number is a palindromic number");
            }
    else
                {
                        printf("Given number is not a palindromic number");
                 }
     return 0;
    }
```

Output:

Enter any number: 121

Given number is a palindrome number

43. Write a Program given number whether it is Prime or not?

A prime number is a natural number that has only one and itself as factors. Examples: 2, 3, 13 are prime numbers.

Program:

```
#include <stdio.h>
main()
{
    int n, i, c = 0;
    printf("Enter any number n: \n");
    scanf("%d", &n);
    /*logic*/
    for (i = 1; i <= n; i++)
  {
     if (n % i == 0)
      {
            c++;
       }
   }
        if (c == 2)
         {
```

```
                printf("n is a Prime number");
                 }
                else
                 {
                        printf("n is not a Prime number");
                 }
            return 0;
        }
```

Output:
Enter any number n: 7
n is Prime

44. What is the difference between C-file and H-file?

- There is no technical difference between .c file and .h file. The compiler will happily let you include a .c file, or compile a .hfile directly, if you want to.
- There is, however, a huge cultural difference:
  - Declarations (prototypes) go in .h files. The .h file is the interface to whatever is implemented in the corresponding .c file.
  - Definitions go in .c files. They implement the interface specified in the .h file.
- The difference is that a .h file can (and usually will) be #included into multiple compilation units (.c files). If you define a function in a .h file, it will end up in multiple .o files, and the linker will complain about a multiply defined symbol. That's why definitions should not go in .h files. (Inline functions are the exception.)
- If a function is defined in a .c file, and you want to use it from other .c files, a declaration of that function needs to be available in each of those other .c files. That's why you put the declaration in a .h, and #include that in each of them. You could also repeat the declaration in each .c file, but that leads to lots of code duplication and an unmaintainable mess.
- If a function is defined in a .c file, but you don't want to use it from other .c files, there's no need to declare it in the header. It's essentially an implementation detail of that .c file. In that case, make the function static as well, so it doesn't conflict with identically-named functions in other files.

45. What the static and dynamic libraries?

- **Static and Dynamic Libraries**

  When a C program is compiled, the compiler generates object code. After generating the object code, the compiler also invokes linker. One of the main tasks for linker is to make code of library functions (eg printf(), scanf(), sqrt(), ..etc) available to your program. A linker can accomplish this task in two ways, by copying the code of library function to your object code, or by making some arrangements so that the complete code of library functions is not copied, but made available at run-time.
- **Static Linking and Static Libraries** is the result of the linker making copy of all used library functions to the executable file. Static Linking creates larger binary files, and need more space on disk and main memory. Examples of static libraries (libraries which are statically linked) are, .a files in Linux and .lib files in Windows.
- **Dynamic linking and Dynamic Libraries** Dynamic Linking doesn't require the code to be copied, it is done by just placing name of the library in the binary file. The actual linking happens when the program is run,

when both the binary file and the library are in memory. Examples of Dynamic libraries (libraries which are linked at run-time) are, .so in Linux and .dll in Windows.

46. Create a use case for static and dynamic libraries?

Static and Dynamic Linking of Libraries

- Static and dynamic linking are two processes of collecting and combining multiple object files in order to create a single executable. Linking can be performed at both compile time, when the source code is translated into machine code and load time, when the program is loaded into memory and executed by the loader, and even at run time, by application programs. And, it is performed by programs called linkers. Linkers are also called link editors. Linking is performed as the last step in compiling a program. In this tutorial static and dynamic linking with C modules will be discussed.

What is Linker?

- Linker is system software which plays crucial role in software development because it enables separate compilation. Instead of organizing a large application as one monolithic source file, you can decompose it into smaller, more manageable chunks that can be modified and compiled separately. When you change one of the modules, you simply recompile it and re-link the application, without recompiling the other source files.
- During static linking the linker copies all library routines used in the program into the executable image. This of course takes more space on the disk and in memory than dynamic linking. But static linking is faster and more portable because it does not require the presence of the library on the system where it runs.

- At the other hand, in dynamic linking shareable library name is placed in the executable image, while actual linking takes place at run time when both the executable and the library are placed in memory. Dynamic linking serves the advantage of sharing a single shareable library among multiple programs.
- Linker as a system program takes relocatable object files and command line arguments in order to generate an executable object file. To produce an executable file the Linker has to perform the symbol resolution, and Relocation.
- Note: Object files come in three flavors viz Relocatable, Executable, and Shared. Relocatable object files contain code and data in a form which can be combined with other object files of its kind at compile time to create an executable object file. They consist of various code and data sections. Instructions are in one section, initialized global variables in another section, and uninitialized variables are yet in another section. Executable object files contain binary code and data in a form which can directly be copied into memory and executed. Shared object files are files those can be loaded into memory and linked dynamically, at either load or run time by a linker.
- Through this article, static and dynamic linking will be explained. While linking, the linker complains about missing function definitions, if there is any. During compilation, if compiler does not find a function definition for a particular module, it just assumes that the function is defined in another file, and treats it as an external reference. The compiler does not look at more than one file at a time. Whereas, linker may look at multiple files and seeks references for the modules that were not mentioned. The separate compilation and linking processes reduce the complexity of program and gives the ease to break code into smaller pieces which are better manageable.
- What is Static Linking?
- Static linking is the process of copying all library modules used in the program into the final executable image. This is performed by the linker and it is done as the last step of the compilation process. The linker

combines library routines with the program code in order to resolve external references, and to generate an executable image suitable for loading into memory.

- Let's see static linking by example. Here, we will take a very simple example of adding two integer quantities to demonstrate the static linking process. We will develop an add module and place in a separate add.c file. Prototype of add module will be placed in a separate file called add.h. Code file addDemo.c will be created to demonstrate the linking process.
- To begin with, create a header file add.h and insert the add function signature into that as follows:

```
int add(int, int);
```

Now, create another source code file viz addDemo.c, and insert the following code into that.

```
#include <add.h>
#include <stdio.h>
int main()
{
  int x= 10, y = 20;
  printf("\n%d + %d = %d", x, y, add(x, y));
  return 0;
}
```

Create one more file named add.c that contains the code of add module. Insert the following code into add.c

```
int add(int quant1, int quant2)
{
  return(quant1 + quant2);
}
```

After having created above files, you can start building the executable as follows:

```
[root@host ~]# gcc -I . -c addDemo.c
```

The -I option tells GCC to search for header files in the directory which is specified after it. Here, GCC is asked to look for header files in the current directory along with the include directory. (in Unix like systems, dot(.) is interpreted as current directory).

Note: Some applications use header files installed in /usr/local/include and while compiling them, they usually tell GCC to look for these header files there.

The -c option tells GCC to compile to an object file. The object file will have name as *.o. Where * is the name of file without extension. It will stop after that and won't perform the linking to create the executable.

As similar to the above command, compile add.c to create the object file. It is done by following command.

```
[root@host ~]# gcc -c add.c
```

This time the -I option has been omitted because add.c requires no header files to include. The above command will produce a new file viz add.o. Now the final step is to generate the executable by linking add.o, and addDemo.o together. Execute the following command to generate executable object file.

```
[root@host ~]# gcc -o addDemo add.o addDemo.o
```

Although, the linker could directly be invoked for this purpose by using ld command, but we preferred to do it through the compiler because there are other object files or paths or options, which must be linked in order to get the final executable. And so, ld command has been explained in detail in How to Compile a C Program section.
Now that we know how to create an executable object file from more than one binary object files. It's time to know about libraries. In the following section we will see what libraries are in software development process? How are they created? What is their significance in software development? How are they used, and many things which have made the use of libraries far obvious in software development?

How to Create Static Libraries?

Static and shared libraries are simply collections of binary object files they help during linking. A library contains hundreds or thousands of object files. During the demonstration of addDemo we had two object files viz add.o, and addDemo.o. There might be chances that you would have ten or more object files which have to be linked together in order to get the final executable object file. In those situations you will find yourself enervate, and every time you will have to specify a lengthy list of object files in order to get the final executable object file. Moreover, fenceless object files will be difficult to manage. Libraries solve all these difficulties, and help you to keep the organization of object files simple and maintainable.

Static libraries are explained here, dynamic libraries will be explained along with dynamic linking. Static libraries are bundle of relocatable object files. Usually they have .a extension. To demonstrate the use of static libraries we will extend the addDemo example further. So far we have add.o which contains the binary object code of add function which we used in addDemo. For more explanatory demonstration of use of libraries we would create a new header file heymath.h and will add signatures of two functions add, sub to that.

```
int add(int, int); //adds two integers
int sub(int, int); //subtracts second integer from first
```

Next, create a file sub.c, and add the following code to it. We have add.c already created.

```
int sub(int quant1, int quant2)
{
  return (quant1 - quant2);
}
```

Now compile sub.c as follows in order to get the binary object file.

```
[root@host ~]# gcc -c sub.c
```

Above command will produce binary object file sub.o.

Now, we have two binary object files viz add.o, and sub.o. We have add.o file in working directory as we have created it for previous example. If you have not done this so far then create the add.o from add.c in similar fashion as sub.o has been created. We will now create a static library by collecting both files together. It will make our final executable object file creation job easier and next time we will have not to specify two object files along with addDemo in order to generate the final executable object file. Create the static library libheymath by executing the following command:

```
[root@host ~]# ar rs libheymath.a add.o sub.o
```

The above command produces a new file libheymath.a, which is a static library containing two object files and can be used further as and when we wish to use add, or sub functions or both in our programs.

To use the sub function in addDemo we need to make a few changes in addDemo.c and will recompile it. Make the following changes in addDemo.c.

```
#include <heymath.h>
#include <stdio.h>

int main()
{
  int x = 10, y = 20;
  printf("\n%d + %d = %d", x, y, add(x, y));
  printf("\n%d + %d = %d", x, y, sub(x, y));
  return 0;
}
```

If you see, we have replaced the first statement #include <add.h> by #include <heymath.h>. Because heymath.h now contains the signatures of both add and sub functions and added one more printf statement which is calling the sub function to print the difference of variable x, and y.

Now remove all .o files from working directory (rm will help you to do that). Create addDemo.o as follows:

```
[root@host ~]# gcc -I . -c addDemo.c
```

And link it with heymath.a to generate final executable object file.

```
[root@host ~]# gcc -o addDemo addDemo.o libheymath.a
```

You can also use the following command as an alternate to link the libheymath.a with addDemo.o in order to generate the final executable file.

```
[root@host ~]# gcc -o addDemo -L . addDemo.o -lheymath
```

In above command -lheymath should be read as -l heymath which tells the linker to link the object files contained in lib<library>.a with addDemo to generate the executable object file. In our example this is libheymath.a.

The -L option tells the linker to search for libraries in the following argument (similar to how we did for -I). So, what we created as of now is a static library. But this is not the end; systems use a lot of dynamic libraries as well. It is the right time to discuss them.

What is Dynamic Linking?

Dynamic linking defers much of the linking process until a program starts running. It performs the linking process "on the fly" as programs are executed in the system. During dynamic linking the name of the shared library is placed in the final executable file while the actual linking takes place at run time when both executable file and library are placed in the memory. The main advantage to using dynamically linked libraries is that the size of executable programs is dramatically reduced because each program does not have to store redundant copies of the library functions that it uses. Also, when DLL functions are updated, programs that use them will automatically obtain their benefits.

How to Create and Use Shared Libraries?

A shared library (on Linux) or a dynamic link library (dll on Windows) is a collection of object files. In dynamic linking, object files are not combined with programs at compile time, also, they are not copied permanently into the final executable file; therefore, a shared library reduces the size of final executable.

Shared libraries or dynamic link libraries (dlls) serve a great advantage of sharing a single copy of library among multiple programs, hence they are called shared libraries, and the process of linking them with multiple programs is called dynamic linking.
Shared libraries are loaded into memory by programs when they start. When a shared library is loaded properly, all programs that start later automatically use the already loaded shared library. Following text will demonstrate how to create and use shared library on Linux.
Let's continue with the previous example of add, and sub modules. As you remember we had two object files add.o, and sub.o (compiled from add.c and sub.c) that contain code of add and sub methods respectively. But we will have to recompile both add.c and sub.c again with -fpic or -fPIC option. The -fPIC or -fpic option enable "position independent code" generation, a requirement for shared libraries. Use -fPIC or -fpic to generate code. Which option should be used, -fPIC or -fpic to generate code that is target-dependent. The -fPIC choice always works, but may produce larger code than -fpic. Using -fpic option usually generates smaller and faster code, but will have platform-dependent limitations, such as the number of globally visible symbols or the size of the code. The linker will tell you whether it fits when you create the shared library. When in doubt, I choose -fPIC, because it always works. So, while creating shared library you have to recompile both add.c, and sub.c with following options:

```
[root@host ~]# gcc -Wall -fPIC -c add.c
[root@host ~]# gcc -Wall -fPIC -c sub.c
```

Above commands will produce two fresh object files in current directory add.o, and sub.o. The warning option -Wall enables warnings for many common errors, and should always be used. It combines a large number of other, more specific, warning options which can also be selected individually. For details you can see man page for warnings specified.
Now build the library libheymath.so using the following command.

```
[root@host ~]# gcc -shared -o libheymath.so add.o sub.o
```

This newly created shared library libheymath.so can be used as a substitute of libheymath.a. But to use a shared library is not as straightforward as static library was. Once you create a shared library you will have to install it. And, the simplest approach of installation is to copy the library into one of the standard directories (e.g., /usr/lib) and run ldconfig command.
Thereafter executing ldconfig command, the addDemo executable can be built as follows. I recompile addDemo.c also. You can omit it if addDemo.o is already there in your working directory.

```
[root@host ~]# gcc -c addDemo.c
[root@host ~]# gcc -o addDemo addDemo.o libheymath.so
```

or

```
[root@host ~]# gcc -o addDemo addDemo.o -lheymath
```

You can list the shared library dependencies which your executable is dependent upon.
The ldd <name-of-executable> command does that for you.

```
[root@host ~]# ldd addition
```

```
libheymath.so => /usr/lib/libheymath.so (0x00002b19378fa000)
libc.so.6 => /lib64/libc.so.6 (0x00002b1937afb000)
/lib64/ld-linux-x86-64.so.2 (0x00002b19376dd000)
```

Last Word

In this tutorial we discussed how static and dynamic linking in C is performed for static and shared libraries (Dynamic Link Libraries - DLLs). Hope you have enjoyed reading this tutorial. Please do write us if you have any suggestion/comment or come across any error on this page. Thanks for reading!

47. What is the difference between Object file and Executable file?

- An **executable file** is a complete program that can be run directly by an operating system (in conjunction with shared libraries and system calls). The file generally contains a table of contents, a number of code blocks and data blocks, ancillary data such as the memory addresses at which different blocks should be loaded, which shared libraries are needed, the entry point address, and sometimes a symbol table for debugging. An operating system can run an executable file more or less by loading blocks of code and data into memory at the indicated addresses and jumping to it.
- Most programs are written with source code logically divided into multiple source files. Each source file is compiled independently into a corresponding "object" file of partially-formed machine code known as object code. At a later time these "object files" are "linked" together to form an executable file.
- **Object files** have a lot in common with executable files (table of contents, blocks of machine instructions and data, and debugging information). However, the code isn't ready to run. It is full of incomplete references to subroutines outside itself, and as such, many of the machine instructions have only placeholder addresses.
- The linker, as a final phase of compilation, will read all of the object files, resolve references between them, perform the final code layout in memory that determines the addresses for all the blocks of code and data, fix up all the placeholder addresses with real addresses, and write out the executable file.

48. Find the size of a variable without using size operator?

To find size of integer data type without using sizeof operator:

```
#include<stdio.h>
int main()
{
  int *ptr = 0;
  ptr++;
  printf("Size of int data type:  %d",ptr);
  return 0;
}
```

To find size of double data type without using sizeof operator:

```
#include<stdio.h>
int main()
{
  double *ptr = 0;
  ptr++;
  printf("Size of double data type:  %d",ptr);
  return 0;
```

```
}
```

To find size of structure data type without using sizeof operator:

```
#include<stdio.h>
struct student
{
int roll;
char name[100];
float marks;
};

int main()
{
 struct student *ptr = 0;
 ptr++;
 printf("Size of the structure student:  %d",ptr);
 return 0;
}
```

To find size of union data type without using sizeof operator:

```
#include<stdio.h>
union student
{
   int roll;
   char name[100];
   float marks;
};
int main()
{
union student *ptr = 0;
ptr++;
printf("Size of the union student:  %d",ptr);
return 0;
}
```

49. Why cannot we declare register storage class as Global variable?

- We cannot use register allocation for global variables because memory is allocated to the global variable at the beginning of the program execution. At that time, it is not certain which function is invoked and which register is used. Function code may use the register internally, but it also has access to a global variable, which might also use the same register. This leads to contradiction, so global register variables are not allowed.
- You can only apply the register specifier to local variables and to the formal parameters in a function. The register specifier also can be used only with variables of block scope. It puts a variable into the register storage class, which amounts to a request that the variable be stored in a register for faster access. It also prevents you from taking the address of the variable.

50. Explain about GDB (GNU Debugger) and BDG?

GNU Debugger:

let us discuss how to debug a c program using gdb debugger in 6 simple steps.

Write a sample C program with errors for debugging purpose

To learn C program debugging, let us create the following C program that calculates and prints the factorial of a number. However this C program contains some errors in it for our debugging purpose.

$ vim factorial.c

```
# include <stdio.h>
int main()
{
    int i, num, j;
    printf ("Enter the number: ");
    scanf ("%d", &num );
    for (i=1; i<num; i++)
        j=j*i;
    printf("The factorial of %d is %d\n",num,j);
}
```

$ cc factorial.c

$ ./a.out

```
Enter the number: 3
The factorial of 3 is 12548672
```

Let us debug it while reviewing the most useful commands in gdb.

Step 1. Compile the C program with debugging option -g

Compile your C program with -g option. This allows the compiler to collect the debugging information.

$ cc -g factorial.c

Note: The above command creates a.out file which will be used for debugging as shown below.

Step 2. Launch gdb

Launch the C debugger (gdb) as shown below.

$ gdb a.out

Step 3. Set up a break point inside C program

Syntax:

break line_number

Other formats:

break [file_name]:line_number

break [file_name]:func_name

Places break point in the C program, where you suspect errors. While executing the program, the debugger will stop at the break point, and gives you the prompt to debug.

So before starting up the program, let us place the following break point in our program.

```
break 10
Breakpoint 1 at 0x804846f: file factorial.c, line 10.
```

Step 4. Execute the C program in gdb debugger

run [args]

- You can start running the program using the run command in the gdb debugger. You can also give command line arguments to the program via run args. The example program we used here does not requires any command line arguments so let us give run, and start the program execution.

run

- Starting program: *$./a.out*
- Once you executed the C program, it would execute until the first break point, and give you the prompt for debugging.

```
Breakpoint 1, main () at factorial.c:10
10                      j=j*i;
```

- You can use various gdb commands to debug the C program as explained in the sections below.

Step 5. Printing the variable values inside gdb debugger

Syntax: print {variable}

Examples:

```
print i
print j
print num

(gdb) p i
$1 = 1
(gdb) p j
$2 = 3042592
(gdb) p num
$3 = 3
(gdb)
```

- As you see above, in the factorial.c, we have not initialized the variable j. So, it gets garbage value resulting in a big numbers as factorial values.
- Fix this issue by initializing variable j with 1, compile the C program and execute it again.
- Even after this fix there seems to be some problem in the factorial.c program, as it still gives wrong factorial value.
- So, place the break point in 10th line, and continue as explained in the next section.

Step 6. Continue, stepping over and in – gdb commands

There are three kind of gdb operations you can choose when the program stops at a break point. They are continuing until the next break point, stepping in, or stepping over the next program lines.

- c or continue: Debugger will continue executing until the next break point.
- n or next: Debugger will execute the next line as single instruction.
- s or step: Same as next, but does not treats function as a single instruction, instead goes into the function and executes it line by line.

By continuing or stepping through you could have found that the issue is because we have not used the <= in the 'for loop' condition checking. So changing that from < to <= will solve the issue.

gdb command shortcuts

Use following shortcuts for most of the frequent gdb operations.

- l – list
- p – print
- c – continue
- s – step
- ENTER: pressing enter key would execute the previously executed command again.

Miscellaneous gdb commands

l command: Use gdb command l or list to print the source code in the debug mode. Use l line-number to view a specific line number (or) l function to view a specific function.

bt: backtrack – Print backtrace of all stack frames, or innermost COUNT frames.

help – View help for a particular gdb topic — help TOPICNAME.

quit – Exit from the gdb debugger.

51. Difference between compiler and cross compiler?

- A native compiler is one that compiles programs for the same architecture or operating system that it is running on. For instance, a compiler running on an x86 processor and creating x86 binaries. A cross-compiler is one that compiles binaries for architectures other than its own, such as compiling SPARC binaries on a PowerPC processor.
- A cross compiler executes in one environment and generates code for another. A "native compiler" generates code for its own execution environment. For example, Microsoft Visual Studio includes a native compiler. It is used on the Windows platform to create applications that are run on the windows platform. A cross compiler could also execute on the Windows operating system, but possibly generate code aimed at a different platform. Many embedded devices, such as mobile phones or washing machines, are programed in such way. Compilers generating cross-platform hyper code such as compilers for Java or any of the .NET languages fall somewhere in between these two basic compiler categories. Their nature depends on the exact use-case, and the angle under which you look at those when categorizing.

52. How to Use C Macros and C Inline Functions with C Code Examples

- **The Concept of C Macros**

  Macros are generally used to define constant values that are being used repeatedly in program. Macros can even accept arguments and such macros are known as function-like macros. It can be useful if tokens are concatenated into code to simplify some complex declarations. Macros provide text replacement functionality at pre-processing time.

  Here is an example of a simple macro :

```
#define MAX_SIZE 10
```

  The above macro (MAX_SIZE) has a value of 10.

  Now let's see an example through which we will confirm that macros are replaced by their values at pre-processing time. Here is a C program :

```
#include<stdio.h>
#define MAX_SIZE 10
int main(void)
                    {
                      int size = 0;
                      size = size + MAX_SIZE;
                      printf("\n The value of size is [%d]\n",size);
                      return 0;
                    }
```

Now lets compile it with the flag -save-temps so that pre-processing output (a file with extension .i) is produced along with final executable :

$ gcc -Wall -save-temps macro.c -o macro

The command above will produce all the intermediate files in the gcc compilation process. One of these files will be macro.i. This is the file of our interest. If you open this file and get to the bottom of this file :

```
...
...
...
int main(void)
{
  int size = 0;
  size = size + 10;

  printf("\n The value of size is [%d]\n",size);

  return 0;
}
```

So you see that the macro MAX_SIZE was replaced with it's value (10) in preprocessing stage of the compilation process.

Macros are handled by the pre-compiler, and are thus guaranteed to be inlined. Macros are used for short operations and it avoids function call overhead. It can be used if any short operation is being done in program repeatedly. Function-like macros are very beneficial when the same block of code needs to be executed multiple times.

Here are some examples that define macros for swapping numbers, square of numbers, logging function, etc.

```
#define SWAP(a,b)({a ^= b; b ^= a; a ^= b;})
#define SQUARE(x) (x*x)
#define TRACE_LOG(msg) write_log(TRACE_LEVEL, msg)
```

Now, we will understand the below program which uses macro to define logging function. It allows variable arguments list and displays arguments on standard output as per format specified.

```
#include <stdio.h>
#define TRACE_LOG(fmt, args...) fprintf(stdout, fmt, ##args);
int main() {
int i=1;
TRACE_LOG("%s", "Sample macro\n");
TRACE_LOG("%d %s", i, "Sample macro\n");
return 0;
}
```

Here is the output:

```
$ ./macro2
Sample macro
1 Sample macro
```

Here, TRACE_LOG is the macro defined. First, character string is logged by TRACE_LOG macro, then multiple arguments of different types are also logged as shown in second call of TRACE_LOG macro. Variable arguments are supported with the use of "…" in input argument of macro and ##args in input argument of macro value.

2. C Conditional Macros

Conditional macros are very useful to apply conditions. Code snippets are guarded with a condition checking if a certain macro is defined or not. They are very helpful in large project having code segregated as per releases of project. If some part of code needs to be executed for release 1 of project and some other part of code needs to be executed for release 2, then it can be easily achieved through conditional macros.

Here is the syntax :

```
#ifdef PRJ_REL_01
..
.. code of REL 01 ..
..
#else
..
.. code of REL 02 ..
..
#endif
```

To comment multiples lines of code, macro is used commonly in way given below :

```
#if 0
..
        .. code to be commented ..
..
#endif
```

Here, we will understand above features of macro through working program that is given below.

```
#include <stdio.h>

        int main()
{

#if 0
        printf("commented code 1");
        printf("commented code 2");
#endif

#define TEST1 1
#ifdef TEST1
        printf("MACRO TEST1 is defined\n");
#endif
#ifdef TEST3
        printf("MACRO TEST3 is defined\n");
#else
        printf("MACRO TEST3 is NOT defined\n");
#endif

        return 0;
}
```

Output:
$ ./macro
MACRO TEST1 is defined
MACRO TEST3 is NOT defined
Here, we can see that “commented code 1”, “commented code 2” are not printed because these lines of code are commented under #if 0 macro. And, TEST1 macro is defined so, string “MACRO TEST1 is defined” is printed and since macro TEST3 is not defined, so “MACRO TEST3 is defined” is not printed.

2. The Concept of C Inline Functions

Inline functions are those functions whose definition is small and can be substituted at the place where its function call is made. Basically they are inlined with its function call.
Even there is no guarantee that the function will actually be inlined. Compiler interprets the inline keyword as a mere hint or request to substitute the code of function into its function call. Usually people say that having an inline function increases performance by saving time of function call overhead (i.e. passing arguments variables, return address, return value, stack mantle and its dismantle, etc.) but whether an inline function serves your purpose in a positive or in a negative way depends purely on your code design and is largely debatable.
Compiler does inlining for performing optimizations. If compiler optimization has been disabled, then inline functions would not serve their purpose and their function call would not be replaced by their function definition.
To have GCC inline your function regardless of optimization level, declare the function with the “always_inline” attribute:

```
void func_test() __attribute__((always_inline));
```

Inline functions provides following advantages over macros.

- Since they are functions so type of arguments is checked by the compiler whether they are correct or not.
- There is no risk if called multiple times. But there is risk in macros which can be dangerous when the argument is an expression.
- They can include multiple lines of code without trailing backlashes.
- Inline functions have their own scope for variables and they can return a value.
- Debugging code is easy in case of Inline functions as compared to macros.
- It is a common misconception that inlining always equals faster code. If there are many lines in inline function or there are more function calls, then inlining can cause wastage of space.

Now, we will understand how inline functions are defined. It is very simple. Only, we need to specify “inline” keyword in its definition. Once you specify “inline” keyword in its definition, it request compiler to do optimizations for this function to save time by avoiding function call overhead. Whenever calling to inline function is made, function call would be replaced by definition of inline function.

```
#include <stdio.h>

void inline test_inline_func1(int a, int b) {
   printf ("a=%d and b=%d\n", a, b);
}
```

```
int inline test_inline_func2(int x) {
   return x*x;
}

int main() {

   int tmp;

   test_inline_func1(2,4);
   tmp = test_inline_func2(5);

   printf("square val=%d\n", tmp);

   return 0;
}
```

Output:

```
$ ./inline
a=2 and b=4
square val=25
```