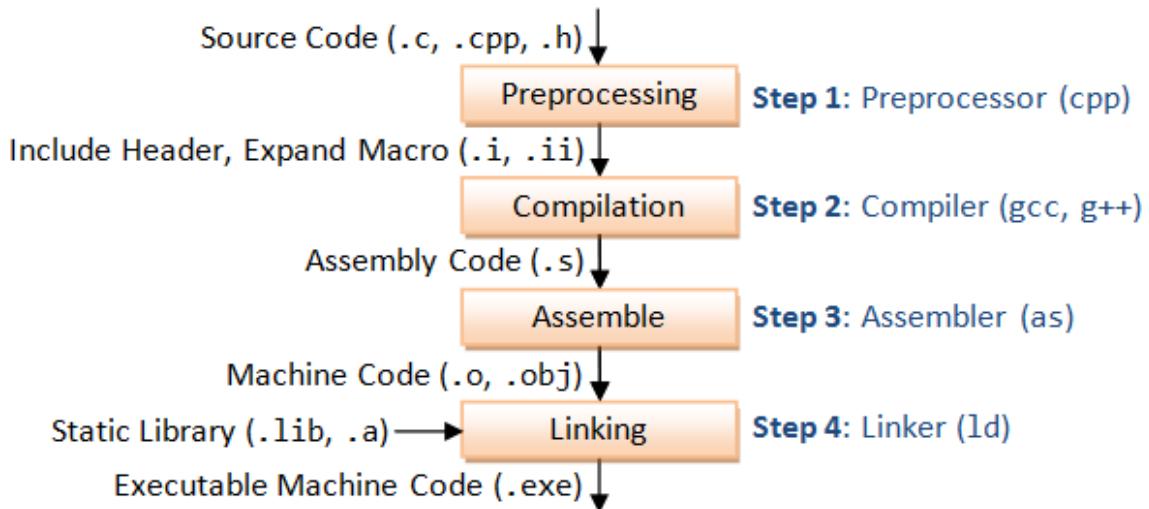


1. Explain about compilation process in C.

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 your 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 into 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

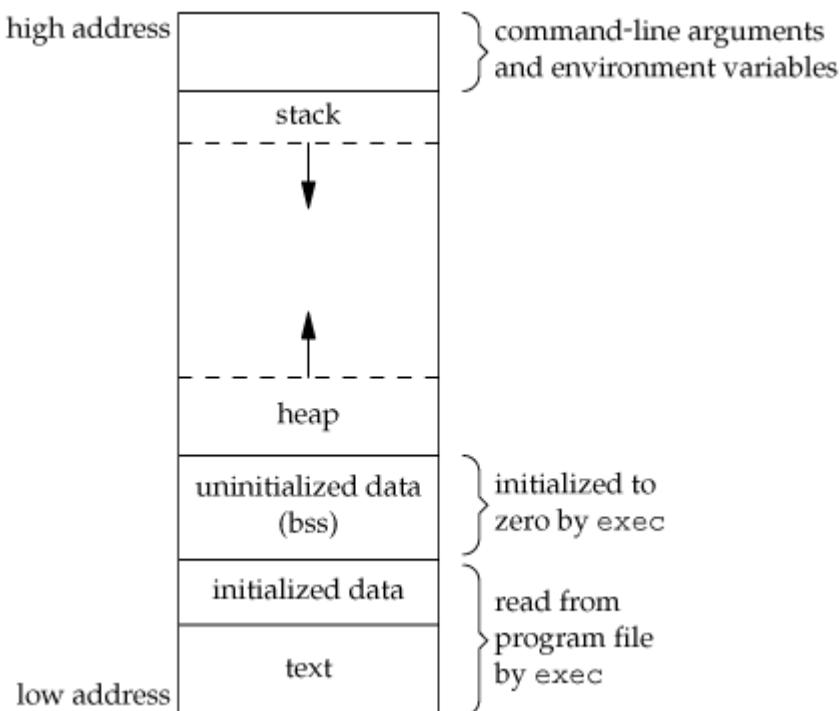
- Once the object file containing the machine code is produced in the step above, the linking step makes sure that all the undefined symbols in code are resolved. An undefined symbol is one for which there is no definition available.
- For example, in a code, there is no definition of printf () function. 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 (bss)
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 naive 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 we add 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 variables 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",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 )
{
    return x > y ? x : y ;
}
...
printf( "%i \n" , max (20 , 10) ) ;
```

When max is "called", the compiler may simply substitute this code.

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

Many compilers can optimize this so that it doesn't require the 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 whereas #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 whereas #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 means 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 program 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 an existing data type. The user-defined data type identifier can be used further to declare variables. It has the following syntax;

Code:

```
typedef datatype identifier;
```

where datatype refers to existing 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 values enclosed 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 all enumeration 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&0xAAAAAAA;
    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

- Uses:

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;
}

```

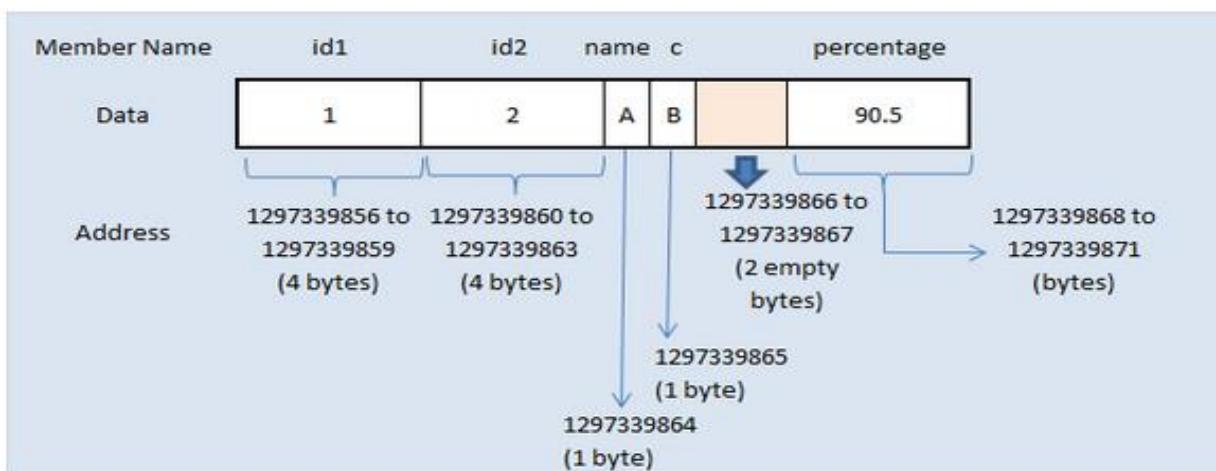
Output:

```

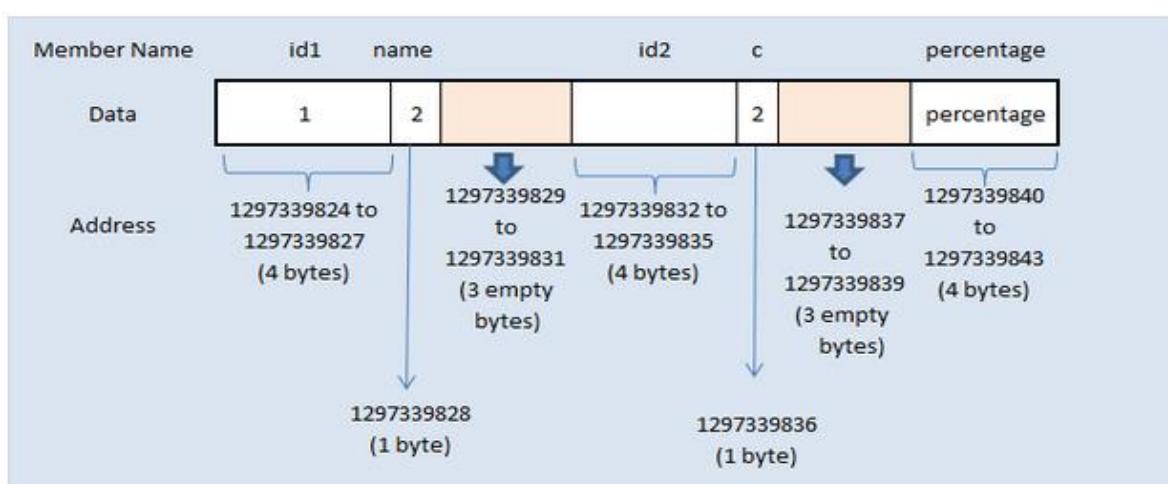
size of structure1 in bytes : 16
Address of id1      = 1297339856
Address of id2      = 1297339860
Address of name     = 1297339864
Address of c        = 1297339865
Address of percentage = 1297339868
sizeof structure2 in bytes : 20
Address of id1      = 1297339824
Address of name     = 1297339828
Address 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 have any technique to allocate 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 returned 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,"malloc()" );
    }
    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 functionsC 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
int main()
{
    int a = 4, b = 7;
    func(&a,&b);      // function calling
}

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

i. Declare function with array as an arguments

Same as f answer

j. Declare structure of arrays

No Answer

k. 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

I. 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

```

m. 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 dereferencing 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.

n. 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"

• Explaination

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 seen in dist3 variable in main function.

o. Structure of functions

No Answer

p. Constant pointer**Constant Pointers**

A constant pointer is a pointer that cannot change the address it is 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;
}
```

q. 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;
}
```

r. Pointer to Constant pointer

No Answer

s. 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;
}
```

t. 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));
```

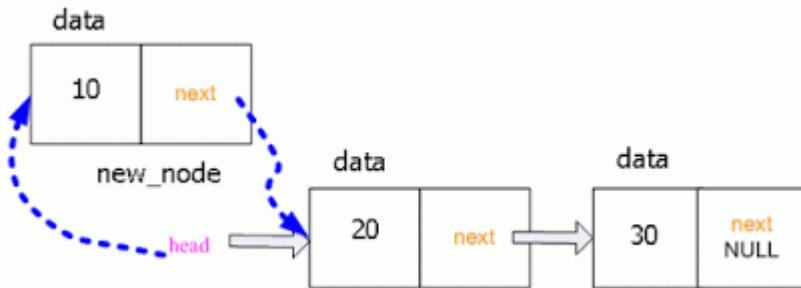
```
    if(new_node == NULL)
        printf("nFailed to Allocate Memory");

    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 :

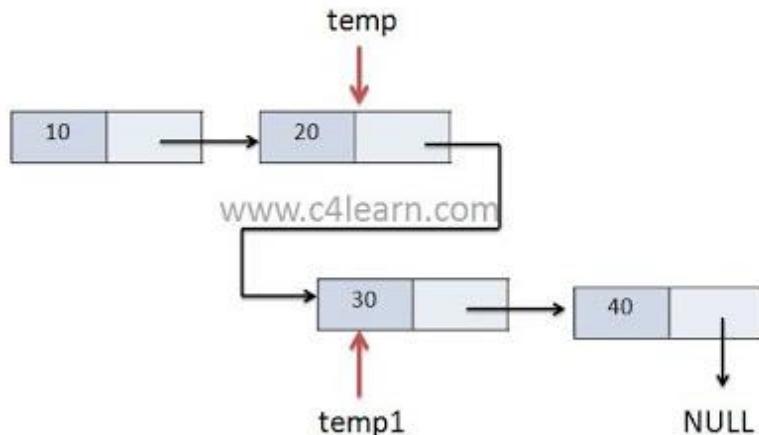
Step 1 : Get Current Position Of "temp" and "temp1" Pointer.

```
temp = start;
```

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

temp1 = temp->next

www.c4learn.com

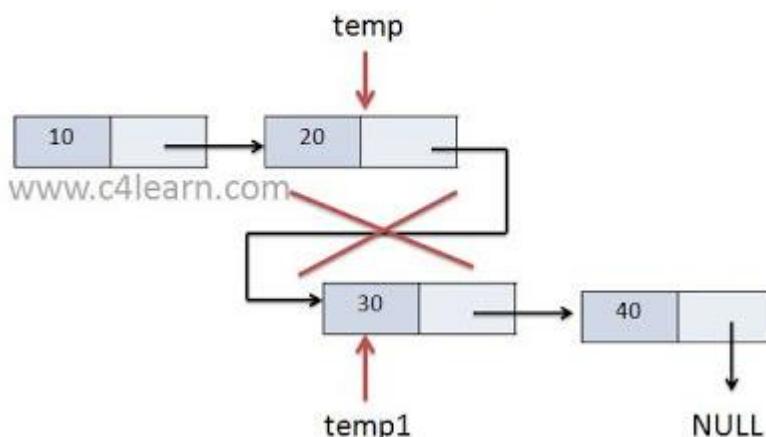


Step 2 :

```
temp1=temp->next;
```

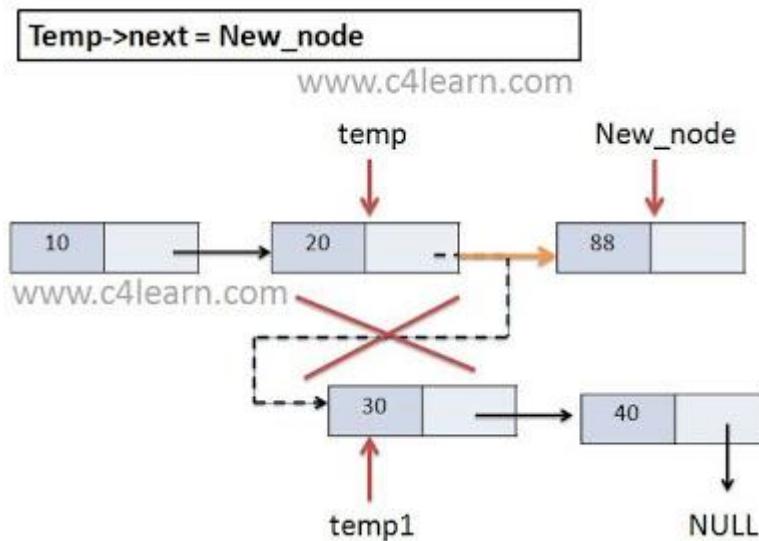
Remove Link Between temp and temp1

www.c4learn.com



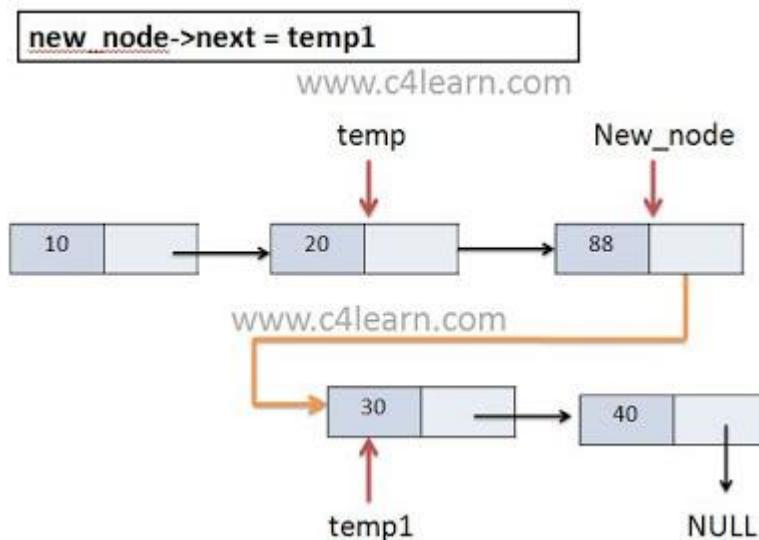
Step 3 :

```
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;

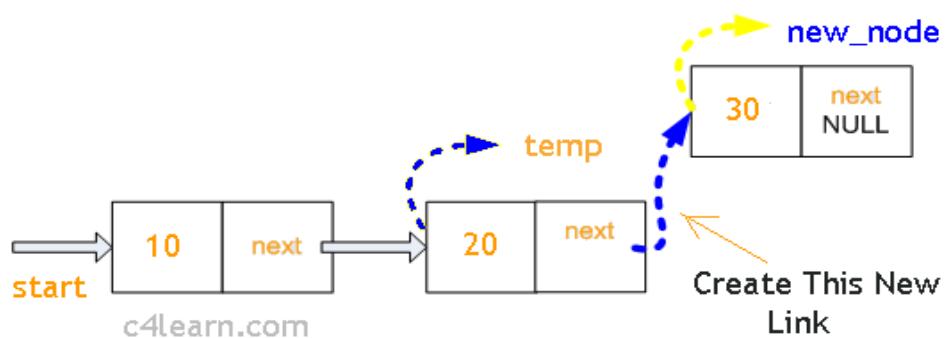
 new_node=(struct node *)malloc(sizeof(struct node));

 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 :

1. 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;
```

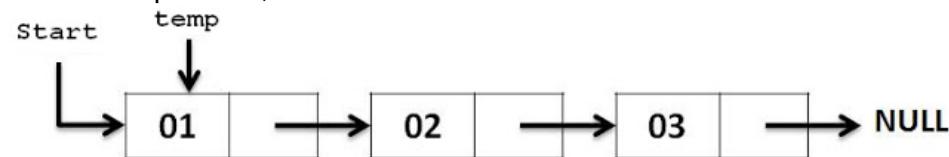
```
temp = start;
start = start->next;
```

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

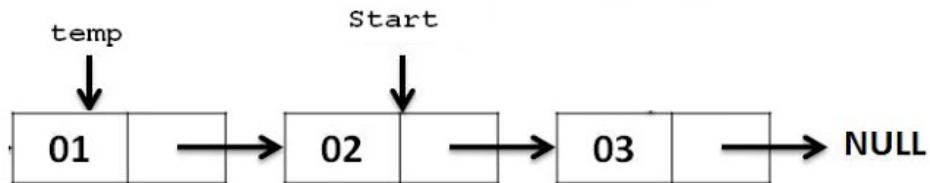
Attention :

Step 1 : Store Current Start in Another Temporary Pointer

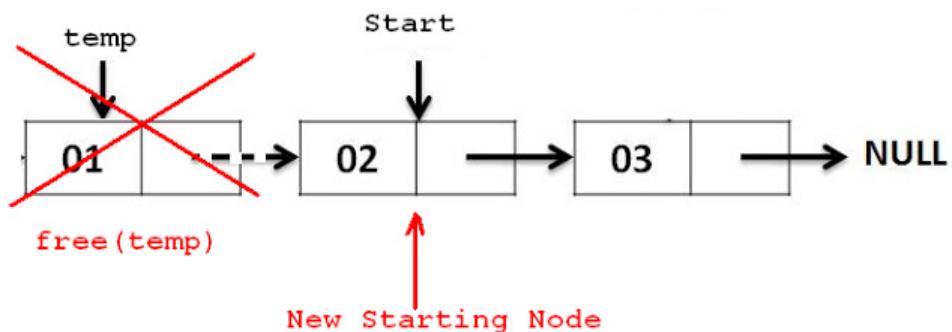
```
temp = start;
```



Step 2 : Move Start Pointer One position Ahead  
 $\text{start} = \text{start}->\text{next};$



Step 3 : Delete temp i.e Previous Starting Node as we have Updated Version of Start Pointer  
 $\text{free}(\text{temp});$



- g. Delete a Node in the mid of the list.
  - h. 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 work 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 of the same type: it is a <b>homogeneous</b> data structure. | Structure is a collection of a fixed number of components of different types. A struct is typically <b>heterogeneous</b> .                 |
| 7.    | Static memory allocation.<br>It uses the subscript to access the array elements.                                       | Dynamic memory allocation.<br>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 timer's 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 handle 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 handle 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>

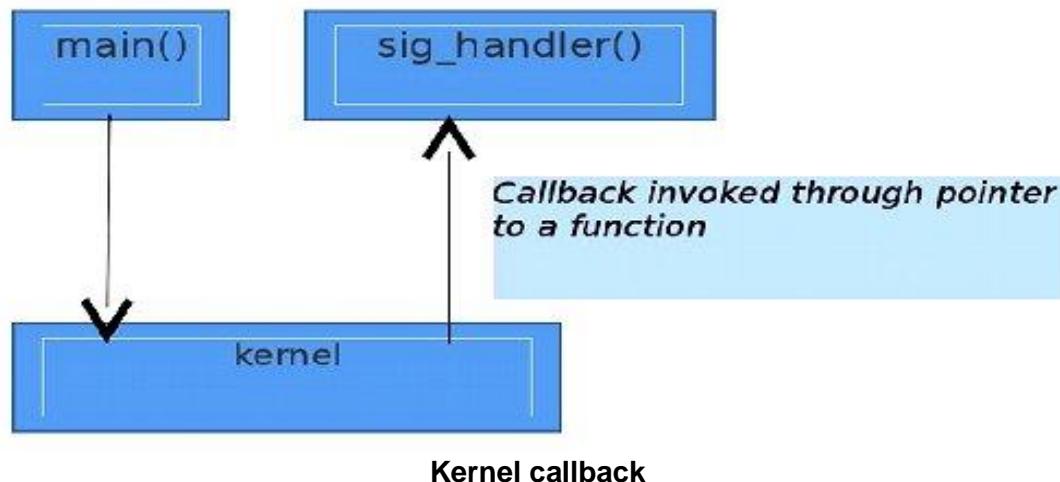
struct sigaction act;

/* 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.



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.

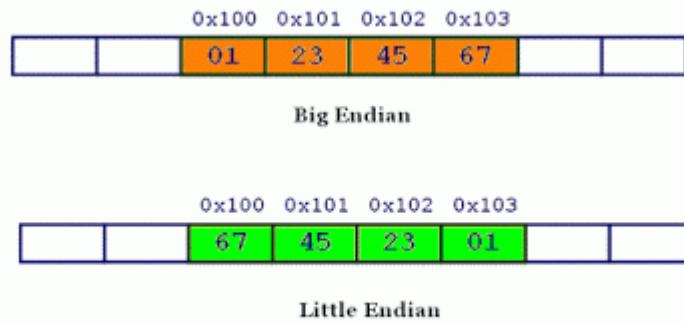
So what can you do about those areas?

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 **0x1234567** 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 of is garbage collected, the struct effectively disappears from unreachable memory as well.

#### 37. Implement following without using library functions

- $\sin(x)$

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

Sine series :  $\sin x = x - ((x^3)/3!) + ((x^5)/5!) - \dots$

Cosine Series :  $\cos x = 1 - ((x^2)/2!) + ((x^4)/4!) - \dots$

Exponential Series :  $e^x = 1 + (x/1!) + ((x^2)/2!) + ((x^3)/3!) + \dots$

- 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 as 1 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 strcpy(char *str1, char *str2);
void main()
{
 char *str1, *str2;
 clrscr();
 printf("\n\n\t ENTER A STRING...: ");
 gets(str1);
 strcpy(str1,str2);
 printf("\n\t THE COPIED STRING IS....: ");
 puts(str2);
 getch();
}
void strcpy(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;
}

```

Method2C 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. **strcmp( )**

```

int custom_strcmp(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 bsize)
{
 size_t slen = strlen(s);
 char *pend = s + slen;

```

```

if(slen + n >= bsize)
 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()
{

```

```

#define 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 a palindrome 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 .h file 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 addtion
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

**I 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) I 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 programmed 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 -fpreprocessed so that pre-processing output (a file with extension .i ) is produced along with final executable :

```
$ gcc -Wall -fpreprocessed 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 its 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 ..
..
#ifndef
..
.. 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
#ifndef TEST1
 printf("MACRO TEST1 is defined\n");
#endif
#ifndef 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 backslashes.
- 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
```

---

## Embedded C Interview Questions and Answers

op Embedded C programming Interview questions and answers for freshers and experienced on embedded system concepts like RTOS, ISR, processors etc. with best answers.

### 1) What is the use of volatile keyword?

The C's volatile keyword is a qualifier that tells the compiler not to optimize when applied to a variable. By declaring a variable volatile, we can tell the compiler that the value of the variable may change any moment from outside of the scope of the program. A variable should be declared volatile whenever its value could change unexpectedly and beyond the comprehension of the compiler.

In those cases it is required not to optimize the code, doing so may lead to erroneous result and load the variable every time it is used in the program. Volatile keyword is useful for memory-mapped peripheral registers, global variables modified by an interrupt service routine, global variables accessed by multiple tasks within a multi-threaded application.

## 2) Can a variable be both const and volatile?

The const keyword make sure that the value of the variable declared as const can't be changed. This statement holds true in the scope of the program. The value can still be changed by outside intervention. So, the use of const with volatile keyword makes perfect sense.

## 3) Can a pointer be volatile?

If we see the declaration volatile int \*p, it means that the pointer itself is not volatile and points to an integer that is volatile. This is to inform the compiler that pointer p is pointing to an integer and the value of that integer may change unexpectedly even if there is no code indicating so in the program.

## 4) What is size of character, integer, integer pointer, character pointer?

- The sizeof character is 1 byte.
- Size of integer is 4 bytes.
- Size of integer pointer and character is 8 bytes on 64 bit machine and 4 bytes on 32 bit machine.

## 5) What is NULL pointer and what is its use?

The NULL is a macro defined in C. Null pointer actually means a pointer that does not point to any valid location. We define a pointer to be null when we want to make sure that the pointer does not point to any valid location and not to use that pointer to change anything. If we don't use null pointer, then we can't verify whether this pointer points to any valid location or not.

## 6) What is void pointer and what is its use?

The void pointer means that it points to a variable that can be of any type. Other pointers points to a specific type of variable while void pointer is a somewhat generic pointer and can be pointed to any data type, be it standard data type(int, char etc) or user define data type (structure, union etc.). We can pass any kind of pointer and reference it as a void pointer. But to dereference it, we have to type the void pointer to correct data type.

## 7) What is ISR?

An ISR(Interrupt Service Routine) is an interrupt handler, a callback subroutine which is called when a interrupt is encountered.

**8) What is return type of ISR?**

ISR does not return anything. An ISR returns nothing because there is no caller in the code to read the returned values.

**9) What is interrupt latency?**

Interrupt latency is the time required for an ISR responds to an interrupt.

**10) How to reduce interrupt latency?**

Interrupt latency can be minimized by writing short ISR routine and by not delaying interrupts for more time.

**11) Can we use any function inside ISR?**

We can use function inside ISR as long as that function is not invoked from other portion of the code.

**12) Can we use printf inside ISR?**

Printf function in ISR is not supported because printf function is not reentrant, thread safe and uses dynamic memory allocation which takes a lot of time and can affect the speed of an ISR up to a great extent.

**13) Can we put breakpoint inside ISR?**

Putting a break point inside ISR is not a good idea because debugging will take some time and a difference of half or more second will lead to different behavior of hardware. To debug ISR, definitive logs are better.

**14) Can static variables be declared in a header file?**

A static variable cannot be declared without defining it. A static variable can be defined in the header file. But doing so, the result will be having a private copy of that variable in each source file which includes the header file. So it will be wise not to declare a static variable in header file, unless you are dealing with a different scenario.

**15) Is Count Down\_to\_Zero Loop better than Count\_Up\_Loops?**

Count down to zero loops are better. Reason behind this is that at loop termination, comparison to zero can be optimized by the compiler. Most processors have instruction for

comparing to zero. So they don't need to load the loop variable and the maximum value, subtract them and then compare to zero. That is why count down to zero loop is better.

### 16) What are inline functions?

The ARM compilers support inline functions with the keyword `__inline`. These functions have a small definition and the function body is substituted in each call to the inline function. The argument passing and stack maintenance is skipped and it results in faster code execution, but it increases code size, particularly if the inline function is large or one inline function is used often.

### 17) Can include files be nested?

Yes. Include files can be nested any number of times. But you have to make sure that you are not including the same file twice. There is no limit to how many header files that can be included. But the number can be compiler dependent, since including multiple header files may cause your computer to run out of stack memory.

### 18) What are the uses of the keyword static?

Static keyword can be used with variables as well as functions. A variable declared static will be of static storage class and within a function, it maintains its value between calls to that function. A variable declared as static within a file, scope of that variable will be within that file, but it can't be accessed by other files.

Functions declared static within a module can be accessed by other functions within that module. That is, the scope of the function is localized to the module within which it is declared.

### 19) What are the uses of the keyword volatile?

Volatile keyword is used to prevent compiler to optimize a variable which can change unexpectedly beyond compiler's comprehension. Suppose, we have a variable which may be changed from scope out of the program, say by a signal, we do not want the compiler to optimize it. Rather than optimizing that variable, we want the compiler to load the variable every time it is encountered. If we declare a variable volatile, compiler will not cache it in its register.

### 20) What is Top half & bottom half of a kernel?

Sometimes to handle an interrupt, a substantial amount of work has to be done. But it conflicts with the speed need for an interrupt handler. To handle this situation, Linux splits the handler into two parts – *Top half* and *Bottom half*. The top half is the routine that actually responds to the interrupt. The bottom half on the other hand is a routine that is scheduled by the upper half to be executed later at a safer time.

All interrupts are enabled during execution of the bottom half. The top half saves the device data into the specific buffer, schedules bottom half and exits. The bottom half does the rest. This way the top half can service a new interrupt while the bottom half is working on the previous.

**21) Difference between RISC and CISC processor.**

RISC (Reduced Instruction Set Computer) could carry out a few sets of simple instructions simultaneously. Fewer transistors are used to manufacture RISC, which makes RISC cheaper. RISC has uniform instruction set and those instructions are also fewer in number. Due to the less number of instructions as well as instructions being simple, the RISC computers are faster. RISC emphasise on software rather than hardware. RISC can execute instructions in one machine cycle.

CISC (Complex Instruction Set Computer) is capable of executing multiple operations through a single instruction. CISC have rich and complex instruction set and more number of addressing modes. CISC emphasise on hardware rather than software, making it costlier than RISC. It has a small code size, high cycles per second and it is slower compared to RISC.

**22) What is RTOS?**

In an operating system, there is a module called the scheduler, which schedules different tasks and determines when a process will execute on the processor. This way, the multi-tasking is achieved. The scheduler in a Real Time Operating System (RTOS) is designed to provide a predictable execution pattern. In an embedded system, a certain event must be entertained in strictly defined time.

To meet real time requirements, the behaviour of the scheduler must be predictable. This type of OS which have a scheduler with predictable execution pattern is called Real Time OS(RTOS). The features of an RTOS are

- Context switching latency should be short.
- Interrupt latency should be short.
- Interrupt dispatch latency should be short.
- Reliable and time bound inter process mechanisms.
- Should support kernel preemption.

**23) What is the difference between hard real-time and soft real-time OS?**

A Hard real-time system strictly adheres to the deadline associated with the task. If the system fails to meet the deadline, even once, the system is considered to have failed. In case of a soft real-time system, missing a deadline is acceptable. In this type of system, a critical real-time task gets priority over other tasks and retains that priority until it completes.

**24) What type of scheduling is there in RTOS?**

RTOS uses pre-emptive scheduling. In pre-emptive scheduling, the higher priority task can interrupt a running process and the interrupted process will be resumed later.

**25) What is priority inversion?**

If two tasks share a resource, the one with higher priority will run first. However, if the lower-priority task is using the shared resource when the higher-priority task becomes ready, then the higher-priority task must wait for the lower-priority task to finish. In this scenario, even though the task has higher priority it needs to wait for the completion of the lower-priority task with the shared resource. This is called priority inversion.

**26) What is priority inheritance?**

Priority inheritance is a solution to the priority inversion problem. The process waiting for any resource which has a resource lock will have the maximum priority. This is priority inheritance. When one or more high priority jobs are blocked by a job, the original priority assignment is ignored and execution of critical section will be assigned to the job with the highest priority in this elevated scenario. The job returns to the original priority level soon after executing the critical section.

**27) How many types of IPC mechanism you know?**

Different types of IPC mechanism are -

- Pipes
- Named pipes or FIFO
- Semaphores
- Shared memory
- Message queue
- Socket

**28) What is semaphore?**

Semaphore is actually a variable or abstract data type which controls access to a common resource by multiple processes. Semaphores are of two types -

- Binary semaphore – It can have only two values (0 and 1). The semaphore value is set to 1 by the process in charge, when the resource is available.
- Counting semaphore – It can have value greater than one. It is used to control access to a pool of resources.

**29) What is spin lock?**

If a resource is locked, a thread that wants to access that resource may repetitively check whether the resource is available. During that time, the thread may loop and check the resource without doing any useful work. Such a lock is termed as spin lock.

### 30) What is difference between binary semaphore and mutex?

The differences between binary semaphore and mutex are as follows -

- Mutual exclusion and synchronization can be used by binary semaphore while mutex is used only for mutual exclusion.
- A mutex can be released by the same thread which acquired it. Semaphore values can be changed by other thread also.
- From an ISR, a mutex can not be used.
- The advantage of semaphores is that, they can be used to synchronize two unrelated processes trying to access the same resource.
- Semaphores can act as mutex, but the opposite is not possible.

### 31) What is virtual memory?

Virtual memory is a technique that allows processes to allocate memory in case of physical memory shortage using automatic storage allocation upon a request. The advantage of the virtual memory is that the program can have a larger memory than the physical memory. It allows large virtual memory to be provided when only a smaller physical memory is available. Virtual memory can be implemented using paging.

A paging system is quite similar to a paging system with swapping. When we want to execute a process, we swap it into memory. Here we use a lazy swapper called pager rather than swapping the entire process into memory. When a process is to be swapped in, the pager guesses which pages will be used based on some algorithm, before the process is swapped out again. Instead of swapping whole process, the pager brings only the necessary pages into memory. By that way, it avoids reading in unnecessary memory pages, decreasing the swap time and the amount of physical memory.

### 32) What is kernel paging?

Paging is a memory management scheme by which computers can store and retrieve data from the secondary memory storage when needed in to primary memory. In this scheme, the operating system retrieves data from secondary storage in same-size blocks called pages. The paging scheme allows the physical address space of a process to be non continuous. Paging allows OS to use secondary storage for data that does not fit entirely into physical memory.

### 33) Can structures be passed to the functions by value?

Passing structure by its value to a function is possible, but not a good programming practice. First of all, if we pass the structure by value and the function changes some of those values, then the value change is not reflected in caller function. Also, if the structure is big, then passing the structure by value means copying the whole structure to the function argument stack which can slow the program by a significant amount.

**34) Why cannot arrays be passed by values to functions?**

In C, the array name itself represents the address of the first element. So, even if we pass the array name as argument, it will be passed as reference and not its address.

**35) Advantages and disadvantages of using macro and inline functions?**

The advantage of the macro and inline function is that the overhead for argument passing and stuff is reduced as the function are in-lined. The advantage of macro function is that we can write type insensitive functions. It is also the disadvantage of macro function as macro functions can't do validation check. The macro and inline function also increases the size of the executable.

**36) What happens when recursive functions are declared inline?**

Inlining an recursive function reduces the overhead of saving context on stack. But, inline is merely a suggestion to the compiler and it does not guarantee that a function will be inlined. Obviously, the compiler won't be able to inline a recursive function infinitely. It may not inline it at all or it may inline it, just a few levels deep.

**37) #define cat(x,y) x##y concatenates x to y. But cat(cat(1,2),3) does not expand but gives preprocessor warning. Why?**

The cat(x, y) expands to x##y. It just pastes x and y. But in case of cat(cat(1,2),3), it expands to cat(1,2)##3 instead of 1##2##3. That is why it is giving preprocessor warning.

**38) ++\*ip increments what?**

It increments the value to which ip points to and not the address.

**39) Declare a manifest constant that returns the number of seconds in a year using preprocessor? Disregard leap years in your answer.**

The correct answer will be -

```
#define SECONDS_IN_YEAR (60UL * 60UL * 24UL * 365UL)
```

Do not forget to use UL, since the output will be very big integer.

**40) Using the variable a, write down definitions for the following:**

- An integer
- A pointer to an integer
- A pointer to a pointer to an integer
- An array of ten integers

- An array of ten pointers to integers
- A pointer to an array of ten integers
- A pointer to a function that takes an integer as an argument and returns an integer
- Pass an array of ten pointers to a function that take an integer argument and return an integer.

The correct answer is as follows -

- int a;
- int \*a;
- int \*\*a;
- int a[10];
- int \*a[10];
- int (\*a)[10];
- int (\*a)(int);
- int (\*a[10])(int);

If you have problem understanding these, please read pointer section in the provide tutorial thoroughly.

**41) Consider the two statements below and point out which one is preferred and why?**

```
#define B struct A *
typedef struct A * C;
```

The typedef is preferred. Both statements declare pointer to struct A to something else and in one glance both looks fine. But there is one issue with the define statement. Consider a situation where we want to declare p1 and p2 as pointer to struct A. We can do this by -

```
C p1, p2;
```

But doing - B p1, p2, it will be expanded to struct A \* p1, p2. It means that p1 is a pointer to struct A but p2 is a variable of struct A and not a pointer.

**42) What will be the output of the following code fragment?**

```
char *ptr;
if ((ptr = (char *)malloc(0)) == NULL)
{
 puts("Got a null pointer");
```

```
}
```

else

```
{
```

```
 puts("Got a valid pointer");
```

```
}
```

The output will be “Got a valid pointer”. It is because malloc(0) returns a valid pointer, but it allocates size 0. So this pointer is of no use, but we can use this free pointer and the program will not crash.

#### 43) What is purpose of keyword const?

The const keyword when used in c means that the value of the variable will not be changed. But the value of the variable can be changed using a pointer. The const identifier can be used like this -

```
const int a; or int const a;
```

Both means the same and it indicates that a is an constant integer. But if we declare something like this -

```
const int *p
```

then it does not mean that the pointer is constant but rather it is pointing to an constant integer. The declaration of an const pointer to a non-constant integer will look like this -

```
int * const p;
```

#### 44) What do the following declarations mean?

```
const int a;
```

```
int const a;
```

```
const int *a;
```

```
int * const a;
```

```
int const * a const;
```

- The first two means that a is a constant integer.
- The third declaration means that a is a pointer to a constant integer.
- The fourth means that a is a constant pointer to a non-constant integer.
- The fifth means that a is a constant pointer to a constant integer.

### 45) How to decide whether given processor is using little endian format or big endian format ?

The following program can find out the endianness of the processor.

```
#include<stdio.h>

main ()
{
 union Test
 {
 unsigned int i;
 unsigned char c[2];
 };
 union Test a = {300};
 if((a.c [0] == 1) && (a.c [1] == 44))
 {
 printf ("BIG ENDIAN\n");
 }
 else
 {
 printf ("LITTLE ENDIAN\n");
 }
}
```

### 46) What is the concatenation operator?

The Concatenation operator (##) in macro is used to concatenate two arguments. Literally, we can say that the arguments are concatenated, but actually their value are not concatenated. Think it this way, if we pass A and B to a macro which uses ## to concatenate those two, then the result will be AB. Consider the example to clear the confusion-

```
#define SOME_MACRO(a, b) a##b

main()
{
 int var = 15;
 printf("%d", SOME_MACRO(v, ar));
}
```

Output of the above program will be 15.

**47) Infinite loops often arise in embedded systems. How does you code an infinite loop in C?**

There are several ways to code an infinite loop -

```
while(1)
```

```
{ }
```

or,

```
for(;;)
```

```
{ }
```

or,

Loop:

```
goto Loop
```

But many programmers prefer the first solution as it is easy to read and self-explanatory, unlike the second or the last one.

**48) Guess the output:**

```
main()
{
 fork();
 fork();
 fork();
 printf("hello world\n");
}
```

It will print "hello world" 8 times. The main() will print one time and creates 3 children, let us say Child\_1, Child\_2, Child\_3. All of them printed once. The Child\_3 will not create any child. Child2 will create one child and that child will print once. Child\_1 will create two children, say Child\_4 and Child\_5 and each of them will print once. Child\_4 will again create another child and that child will print one time. A total of eight times the printf statement will be executed.

**49) What is forward reference w.r.t. pointers in c?**

Forward Referencing with respect to pointers is used when a pointer is declared and compiler reserves the memory for the pointer, but the variable or data type is not defined to which the pointer points to. For example

```
struct A *p;
struct A
{
 // members
};
```

**50) How is generic list manipulation function written which accepts elements of any kind?**

It can be achieved using void pointer. A list may be expressed by a structure as shown below

```
typedef struct
{
 node *next;
 /* data part */

}node;
```

Assuming that the generic list may be like this

```
typedef struct
{
 node *next;
 void *data;
}node;
```

This way, the generic manipulation function can work on this type of structures.

### 51) How can you define a structure with bit field members?

Bit field members can be declared as shown below

```
struct A
{
 char c1 : 3;
 char c2 : 4;
 char c3 : 1;
};
```

Here c1, c2 and c3 are members of a structure with width 3, 4, and 1 bit respectively. The ':' indicates that they are bit fields and the following numbers indicates the width in bits.

**52) How do you write a function which takes 2 arguments - a byte and a field in the byte and returns the value of the field in that byte?**

The function will look like this -

```
int GetFieldValue(int byte, int field)
{
 byte = byte >> field;
 byte = byte & 0x01;
 return byte;
}
```

The byte is right shifted exactly n times where n is same as the field value. That way, our intended value ends up in the 0th bit position. "Bitwise And" with 1 can get the intended value. The function then returns the intended value.

**53) Which parameters decide the size of data type for a processor ?**

Actually, compiler is the one responsible for size of the data type. But it is true as long as OS allows that. If it is not allowable by OS, OS can force the size.

**54) What is job of preprocessor, compiler, assembler and linker ?**

The preprocessor commands are processed and expanded by the preprocessor before actual compilation. After preprocessing, the compiler takes the output of the preprocessor and the source code, and generates assembly code. Once compiler completes its work, the assembler takes the assembly code and produces an assembly listing with offsets and generate object files.

The linker combines object files or libraries and produces a single executable file. It also resolves references to external symbols, assigns final addresses to functions and variables, and revises code and data to reflect new addresses.

**55) What is the difference between static linking and dynamic linking ?**

In static linking, all the library modules used in the program are placed in the final executable file making it larger in size. This is done by the linker. If the modules used in the program are modified after linking, then re-compilation is needed. The advantage of static linking is that the modules are present in an executable file. We don't want to worry about compatibility issues.

In case of dynamic linking, only the names of the module used are present in the executable file and the actual linking is done at run time when the program and the library modules both are present in the memory. That is why, the executables are smaller in size. Modification of the library modules used does not force re-compilation. But dynamic linking may face compatibility issues with the library modules used.

### 56) What is the purpose of the preprocessor directive #error?

Preprocessor error is used to throw a error message during compile time. We can check the sanity of the make file and using debug options given below

```
#ifndef DEBUG

#ifndef RELEASE

#error Include DEBUG or RELEASE in the makefile

#endif

#endif
```

### 57) On a certain project it is required to set an integer variable at the absolute address 0x67a9 to the value 0xaa55. The compiler is a pure ANSI compiler. Write code to accomplish this task.

This can be achieved by the following code fragment:

```
int *ptr;

ptr = (int *)0x67a9;

*ptr = 0xaa55;
```

### 58) Significance of watchdog timer in Embedded Systems.

The watchdog timer is a timing device with a predefined time interval. During that interval, some event may occur or else the device generates a time out signal. It is used to reset to the original state whenever some inappropriate events take place which can result in system malfunction. It is usually operated by counter devices.

### 59) Why ++n executes faster than n+1?

The expression ++n requires a single machine instruction such as INR to carry out the increment operation. In case of n+1, apart from INR, other instructions are required to load the value of n. That is why ++n is faster.

**60) What is wild pointer?**

A pointer that is not initialized to any valid address or NULL is considered as wild pointer.  
Consider the following code fragment -

```
int *p;

*p = 20;
```

Here p is not initialized to any valid address and still we are trying to access the address. The p will get any garbage location and the next statement will corrupt that memory location.

**61) What is dangling pointer?**

If a pointer is de-allocated or freed and the pointer is not assigned to NULL, then it may still contain that address and accessing the pointer means that we are trying to access that location and it will give an error. This type of pointer is called dangling pointer.

**62) Write down the equivalent pointer expression for referring the same element a[i][j][k][l] ?**

We know that a[i] can be written as \*(a+i). Same way, the array elements can be written like pointer expression as follows -

```
a[i][j] == *(*(a+i)+j)

a[i][j][k] == *(*(*(a+i)+j)+k)

a[i][j][k][l] == *(*(*(*(a+i)+j)+k)+l)
```

**63) Which bit wise operator is suitable for checking whether a particular bit is on or off?**

"Bitwise And" (&) is used to check if any particular bit is set or not. To check whether 5'th bit is set we can write like this

```
bit = (byte >> 4) & 0x01;
```

Here, shifting byte by 4 position means taking 5'th bit to first position and "Bitwise And" will get the value in 0 or 1.

**64) When should we use register modifier?**

The register modifier is used when a variable is expected to be heavily used and keeping it in the CPU's registers will make the access faster.

### 65) Why doesn't the following statement work?

```
char str[] = "Hello" ;
strcat (str, '!') ;
```

The string function strcat( ) concatenates two strings. But here the second argument is '!', a character and that is the reason why the code doesn't work. To make it work, the code should be changed like this:

```
strcat (str, "!") ;
```

### 66) Predict the output or error(s) for the following program:

```
void main()
{
 int const * p = 5;
 printf("%d", ++(*p));
}
```

The above program will result in compilation error stating “Cannot modify a constant value”. Here p is a pointer to a constant integer. But in the next statement, we are trying to modify the value of that constant integer. It is not permissible in C and that is why it will give a compilation error.

### 67) Guess the output:

```
#include<stdio.h>

main()
{
 unsigned int a = 2;
 int b = -10;
```

```
(a + b > 0)?puts("greater than 0"):puts("less than 1");
}
```

*Output - greater than o*

If you have guessed the answer wrong, then here is the explanation for you. The  $a + b$  is -8, if we do the math. But here the addition is between different integral types - one is unsigned int and another is int. So, all the operands in this addition are promoted to unsigned integer type and b turns to a positive number and eventually a big one. The outcome of the result is obviously greater than 0 and hence, this is the output.

### 68) Write a code fragment to set and clear only bit 3 of an integer.

```
#define BIT(n) (0x1 << n)

int a;

void SetBit3()
{
 a |= BIT(3);
}
```

```
void ClearBit3()
{
 a &= ~BIT(3);
}
```

### 69) What is wrong with this code?

```
int square(volatile int *p)
{
 return *p * *p;
}
```

The intention of the above code is to return the square of the integer pointed by the pointer p. Since it is volatile, the value of the integer may have changed suddenly and will result in something else which will look like the result of the multiplication of two different integers. To work as expected, the code needs to be modified like this.

```
int square(volatile int *p)
{
 int a = *p;
 return a*a;
}
```

#### 70) Is the code fragment given below is correct? If so what is the output?

```
int i = 2, j = 3, res;
res = i+++j;
```

The above code is correct, but little bit confusing. It is better not to follow this type of coding style. The compiler will interpret above statement as “res = i++ + j”. So the res will get value 5 and i after this will be 3.

.....  
.....  
.....

#### C++ questions:

- 13.What is copy constructor and a default constructor and write a copy constructor and explain shallow copy with memory diagram.
- 14.Is it necessary to have const for a reference in copy constructor
- 15.Explain the stack over flow condition when pass by value is done in copy constructor.
- 16.Memory layout of a program and explain the different segments when a function call happens.
17. Explain the difference between static variable and local variable
18. what are the different storage classes present and explain the differences between them.
- 19.Polymorphism and virtual concepts
- 20.Why a virtual destructor is needed explain with a scenario what happens when a virtual destructor is not defined.

- 21.What is a smart pointer and why it is needed?
- 22.Difference between smart pointer and auto pointer and write the code for autp ptr implementation.
- 23.exception handling in c++
- 24.what is stack unwinding?
- 25.Difefrence between vector and list.
- 26.Templates and function templates.
- 27.what is operator overloading ? implememt how to overload ++ operator and "insertion" operator.
28. What is memory leak and memory corruption and explain the scenario for memory leaks and memory corruption with a sample code.
- 29.What is factory pattern? why factory pattern is needed?
- 30.What is builder pattern pattern ? write a sample code for the same.
- 31.What is a singleton pattern and write the code for the same.
32. How can you retain the values of a local variable without using a static variable.
- 33.What happens when a function is made inline?
34. why inline functions are needed and in what scenario the inlines are needed?
  
- 35..Find the first non repeated value in the array by iterating only once.

Posted by [RAJASEKHAR CHALLA](#) at 9:07 AM [No comments:](#)

[Email This](#)[Blog This!](#)[Share to Twitter](#)[Share to Facebook](#)[Share to Pinterest](#)

### **Linux Device Driver,Embedded C Interview Questions**

**Below are the collection of interview questions for positions in Embedded,Linux Device Drivers,QNX BSP Kernel Programming, C language,System programmer.**

\*\*\*\*\*

Mirafra Interview Questions for QNX OS for Automotive Domain(Client : Qualcomm)

Round-1

1.Write a macro to swap the nibbles in the word ?

Ex:0x1234 output:2143

2.Differance between macro & micro Kernel.

3.what you observed difference between QNX & Linux

4.Difference between Semaphore & Mutex

5.Write a programme to delete a given node number in Single linked list.

6.difference between processes and thread

7.diffrence b/w RTOS & GPOS

8.code base query :

Need the out put to be printed as 10.

```
int a=10;
main()
{
 int a=20;
 printf("%d",a);
}
```

Round-2

1.What are the projects you worked and breif them

2.What is the difference between I2C & SPI

3.How will you chosse which protocal to be used on board communicaiton

4.How the communication difference b/w I2C & SPI

5.Sceanrio based query:

Had 4 sensors on board,which communicaiton you will chose to establish communicaiton b/w the sensors

6.Code base query:

what is the output.

```
main()
{
```

```
int *p= &a;
int a= 5;
printf("%d %d",*p,a);
}
```

7.Different types of storage classes and details of each storage class

8.Detailed discussion on Static like storage ,default ,scope,life time.

9.Code based query:

what is the output.

```
main()
{
 int a= 5;
 int *p= &a;
 printf("%d %d",*p,a);
}
```

```
main()
{
 int a= 5;
 int *p= &a;
 *p++;
 p--;
 printf("%d %d",*p,a);
}
```

```
main()
{
```

```
int a= 5;
int *p= &a;
(*p)++;
p--;
printf("%d %d",*p,a);
}
```

10.

```
const volatile int *ptr;
*const volatile int ptr;
```

What is the diff .

11.

which pointer value we can modify.

```
const volatile int *ptr;
*const volatile int ptr;
```

12.different IPC mechanisms in QNX & Linux

13.Discussion on Shared Memory

14.Roles & Responsibilities in the project

15.What is grub loader

16.Embedded board Booting sequence

17.Diff b/w MLO & IPL.

18.Who will provide MBR and is it possible to modify.

19.Debugging techniques.

Round-3

1.Why are you leaving the company, what are the reasons.

2. Write a macro to swap the nibbles in the word ?

Ex:0x1234 output:2143

3. Thread locked a resource ,before releasing the resource an Interrupt got triggered ,here interrupt is also using the same resource.

How will you handle this scenario.

4.Having Three resources and 10 Threads,

out of 10 ,3 threads acquired the resources, How will handle the reaming 7 threads

which thread has to come first to take the release resources

Which type of mechanism will you use for synchronization.

5.Difference b/w Semaphore & Mutex /binary sema & Mutex.Is it possible to use the counting semaphore for above scenario

6.What is Bit Binding in I2C protocol

7.Multimaster & single slave communication

m1: 0001 0000

m2: 0101 1010

m3: 0000 1111

Which master will acquire I2C Bus

8.What is i2c clock stretching

9. Is clock stretching is for Master or Slave in I2C protocol

10. How Multiple Threading is handled in single CPU.

### Panasonic America Automotive for BSP Role for QNX OS

1. Is const volatile possible ?

2. Where a const volatile variable is used?

3. What is booting sequence of linux?

4. Booting Sequence of a board ?

5. Why primary boot loader and secondary boot loader required?

6. What are types of boot loader?

7. How a user mode is transferred to kernel mode?

What part of kernel driver worked. Explain

Difference between kernel/user space

How system call causes change from user to kernel space

Which RTOS worked on. Difference between various OS/RTOS

How to chose RTOS for a consumer product

Measure of performance of OS. Define performance

How to debug crash. Gdb

What is object file, how its used in crash

What are the various code optimization techniques used

memory leak deduction and various ways of handling

How to write own malloc call

How to proceed if system is sluggish

How to determine if some high prio task is hogging CPU

|                                                             |
|-------------------------------------------------------------|
| How debug prints/system trace could help solve above issues |
| Important things to look for in code reviews                |
| Experience with Oscilloscope, logic analyzer, i2C sniffer   |
| Understanding of schematics                                 |
| Why driver code in not written in C++/Java                  |
| How to debug while system is running                        |
| Run Time optimization                                       |
| Estimate the requirement in BSP domain                      |
| How to define the process of QNX/Arch delivery integration  |
| How many years of experience in BSP domain                  |

|                                                          |
|----------------------------------------------------------|
| What part of kernel driver worked. Explain               |
| Difference between kernel/user space                     |
| How system call causes change from user to kernel space  |
| Which RTOS worked on. Difference between various OS/RTOS |
| How to chose RTOS for a consumer product                 |
| Measure of performance of OS. Define performance         |
| How to debug crash. Gdb                                  |
| What is object file, how its used in crash               |
| What are the various code optimization techniques used   |
| memory leak deduction and various ways of handling       |
| How to write own malloc call                             |
| How to proceed if system is sluggish                     |
| How to determine if some high prio task is hogging CPU   |

|                                                                                                       |
|-------------------------------------------------------------------------------------------------------|
| How debug prints/system trace could help solve above issues                                           |
| Important things to look for in code reviews                                                          |
| Experrience with Oscilloscope, logic analyzer, i2C sniffer                                            |
| Understanding of schematics                                                                           |
| Why driver code in not written in C++/Java                                                            |
| How to debug while system is running                                                                  |
| Run Time optimization                                                                                 |
| Estimate the requirement in BSP domain                                                                |
| How to define the process of QNX/Arch delivery integration                                            |
| How many years of experience in BSP domain                                                            |
| what is repeate sequence in I2C?<br>How this signal will look on CRO?                                 |
| How many lines required for SPI communication?                                                        |
| Do you need to change Clock polarity and phase for SPI?                                               |
| Who has control of SPI clock?                                                                         |
| What is deadlock?                                                                                     |
| How to come out of deadlock?                                                                          |
| what is difference between mutex and semaphore?                                                       |
| Explain board bring up.                                                                               |
| What is the difference between inline and macro?                                                      |
| Write program toreset bit in register if address of register is given<br>reset (int address, int bit) |
| How do you implement malloc ?                                                                         |
| How to know size of memory allocated by malloc using pointer?                                         |
| What is the problem in following code<br><code>char * p = 'a';</code><br>How to fix it?               |

Explain cinit and bss section

Asked on what are optimisation that I have done in my previous project

how will you verify I2C communication? How will you know if there was a software or hw problem?

How will you know if the I2C lines are noisy?

Have you worked in crash dump?

How will you solve memory crash, what steps would you take if a crash occurs?

Diff between mutex and semaphore? Can semaphore be used for data synchronisation purpose?

Data sharing between ISR and threads?

Whats the boot loading process from powerup? They wanted a detailed answer to this asking for detailed explanation for every step?

what is NAND and NOR flash, diff between them?

What execution in place?

What monitoring tools have you used like I2C sniffer

How many years of experience in BSP domain

Details of performance related optimisation?

### HARMAN Automotive :

\*\*\*\*\*  
\*\*\*\*\*

1.What memcpy do? implement your own memcpy function?with optimization(less usage of variables?)

2. int \*i

    char \*c

    void \*v

tell me the sizes ?

sizeof(\*i)

sizeof(i)

sizeof(\*c)

sizeof(c)

sizeof(\*v)

sizeof(v)

3.What is big endian and little endian?

    implement a function changes the endians

- 4.what is cache coherence?
- 5.what is critical section?
- 6.what is protection mechanism?
- 7.Can a scheduler can be locked?
- 8.How are the compiler optimization techniques?
- 9.what is user space and kernel space?
- 10.Main Advantages and disadvantages of having separate user space and kernel space?
- 11.what is scheduler?
- 12.who schedules the scheduler?
- 13.At what frequency scheduler looks for threads/processes ready for schedule?
- 14.what is tick?
- 15.what actually does a system does in a tick?
- 16.what is DMA?
- 17.When cahce is enabled in a operating system ,DMA is enabled,how does DMA access the cache?
- 18.what is scheduler algoritham?
- 19.How interrupts are generated?
20. write a programe which includes the variables should cover all the segments ?
- 21.Local Const variables reside in which segment?
- 22.Global const variables reside in which segment?
23. How many stack will present?
- 24.does each processes has it own stack ?
- 25.does each thread has its own stack?
- 26.what is context switch ?
- 27.when their context switch from thread t1 To thread t2 where does the stack information of t1 gets stored?
- 28.What is advantage of virtual memory?  
why not directly the physical address?
- 29.What is MMU?
- 30.what is a page fault?
- 31.What is instruction pipelining?
- 32.What is page write back?

1. Memory map of RAM
2. Where do const variables get stored? (literal segment)
3. Static, auto, etc
4. Write a program to change endianness
5. What is structure padding?  
Why does the compiler perform padding?  
How to send information over network? (structure packing)
6. Declare an array of 10 integer pointers
7. Volatile variable in-depth (definition, use case, impact, when to avoid, etc)
8. Where does dynamic memory allocation get stored?  
What are the pros and cons of using dynamic memory allocation versus static memory allocation?  
If you had to allocate 1MB of memory that needs to be used for only a short time, which type of allocation would you use?
9. Different types of schedulers (round robin, pre-emptive)

10. Timeslice, jiffy
11. Priority inversion, priority inheritance, priority ceiling
12. Difference between semaphores & mutexes
13. Write a program to implement memcpy.  
Optimize it such that it uses **NO** stack space at all
14. How to handle situation when a program is too large for code space? (edit linker file)

1.Difference btwn Process and Thread

2.Diff btwn mutex & semaphore

3.Bit operations

4.C program for factorial, prime number with recursion

5.About Booting process

6.Questions on Pointers

7.What is Process preemption

8.About priority inversion & priority inheritance

9.How priority inheritance will work

10.structure padding

11.about #pragma

12.How to avoid re declaration of header files

13.About Jtag

1.How you Evaluate the Driver

2.about ISR, Interrupts

3.about Synchronization mechanism where you used in real time(examples)

4.what happen if you use mutex in ISR, how you over come with that problem

5.about memory mapped files

6.about linker scripts

7.what is re entrant function

8.what is module

9. IPC mechanism

10.toggle alternate bits

- 11.about function pointers
- 12.Advantages of func pointers
- 13.Advantages of call back functions
14. Declare a array of function pointer of taking two arguments
15. How to find little endian and big endian

1. Implement your own memcpy function and Error handling in memcpy
2. How you handle errors with the pointers
3. About dynamic and static memory
- 4.about malloc in depth
5. about Null pointer
6. how can you determine whether your memory is in protected or un-protected mode
7. About memory layout
- 8.How you will think while writing a program in optimized way
- 9.They will ask most of the question from memory in this round please study MMU well.

### Some Imp Question:

- 1.About volatile and use cases
- 2.Why the first two parameters are of type void in memcpy
- 3.Are #pragramas Gerneric or processor specific
- 4.Enabling & Disabling of interrupts
- 5.C program to swap every 2 bits in a 8 bit binary number.
- 6.Write a program to find how many bit to toggle in 2 binary numbers so that they become equal
  - a. program for to get the number of bits toggle in 2 binary numbers and toggle them to make the numbers equal
7. what is user mode and kernel mode

- 1.How you Decide the stack size for the function or thread.
- 2.Booting Sequence

3. About JTAG
  4. Diff btwn SPI & I2C
  5. Memory Layout
  6. what is your strength
  7. About Disassembly file(Memory map file)
  8. About clock & timers
  9. About Stack
- \*\*\*\*\*  
\*\*\*\*\*

### PANASONIC and HARMAN Automotive:

- \*\*\*\*\*  
\*\*\*\*\*
1. Booting sequence in embedded systems
  2. How to decrease the time of booting processes
  3. What is the functionality of PROBE function
  4. How to detect whether a device is not detected?
  5. Own Malloc implementations
  6. memcpy implementation
  7. Find a word from string
  8. How to find if there is in repeating node in linked list
  9. For a given array gather all 1's to one side and 0's to one side.
  10. In a given byte interchange pair of bits
  11. Reverse linked list
  12. Reverse string
  13. Reverse a word in a string

### Ericson:

- \*\*\*\*\*  
\*\*\*\*\*
1. A thread is created by processes, how the process comes to know the completion of that created thread ?
  2. In a big array consists of 1's & 0's , write an efficient programme to keep all 1's to right side and 0's to other side?
  3. Write a programme In a given string find a specific word.
  4. Semaphore internal code behavior in a way semaphore implementation ?
  5. Topic to look :
    1. Inter-process communication
    - a. Socket programming

- b. Named and unnamed pipes
  - c. Message queues
  - d. Shared memory, mmap
2. Synchronization mechanisms
- a. Mutex and semaphore
  - b. Spinlocks
  - c. Synchronization between ISRs and process context threads
3. Debugging using GDB

### C Questions

---

- 1. Storage classes
- 2. const and volatile
- 3. How to allocate memory in c - malloc vs calloc
- 4. brk, sbrk
- 5. Any other system call to allocate memory. mmap ; anonymous mapping
- 6. Memory barriers; Why are they required
- 7. Wild and dangling pointers
- 8. struct alignment and packing. Need of alignment
- 9. Pass by value and ref
- 10. Endianness
- 11. size of void pointer
- 12. What is thread safety
- 13. What is re-entrancy
- 14. far and near pointers

### Programming

---

- 1. Verify if a number is a power of two
- 2. nth element from last in a linked list

### OS Concepts

- 
1. Thrashing
  2. Deadlock - example
  3. MM - Segmentation, paging, swapping
  4. Paging vs swapping
  5. Different segments in a program
  6. Does linux use segmentation
  7. System boot sequence
  8. What is DMA. Modes - cycle stealing/burst (blk transfer)/transparent
  9. Cache coherency during dma. Which component handles it

Linux

- 
1. How are interrupts handled in linux
  2. spinlock vs mutex vs semaphore
  3. Top and bottom halves. examples
  4. Different contexts
  5. softirq vs tasklet vs workqueue
  6. IPC mechanisms - shared mem/pipe/sockets/DBUS
  7. Memory alloc in linux - kmalloc vs vmalloc
  8. System call implementation
  9. Physical Virtual and logical addresses
  10. Threads vs processes

scheduler in linux

Hardware / Protocols / Misc

---

I2C

- Protocol
- Slave address

\*\*\*\*\*  
\*\*\*\*\*

### Dover corporation:F2F

- 1.what is a device driver and write a simple driver and explain what happens when an insmod is done an module
- 2.How will you insert a module statically in to linux kernel.
- 3.what are IPC mechanisms and explain d/f between all types with example scenario
- 4.explain the synchronization mechanisms
- 5.Explain about file system in linux kernel and how a kernel handles any file when it was created.
- 6.what is file descriptor and what information it contains in file descriptor
- 7.How do u see system messages and how do find the issues in system by reading the proc file system or dmsg queues or sysfs information.
- 8.what all kind of info is present in procfs.
- 9.when do u select the semaphore and a mutex and binary sempahore.
- 10.what is an interrupt and difference between exceptions and signals
- 11.How many types of signals are present in linux kernel.

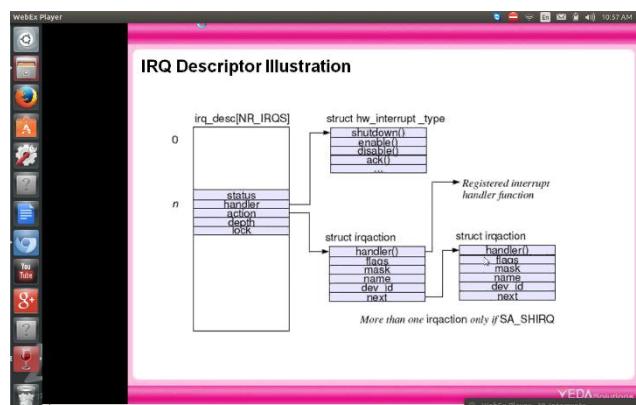
### Shared IRQ :

When a signal is raised on irq line which triggers an exception on processor.

Linux had data structure called interrupt descriptor table which is linked list of irq descriptors number of descriptors is equal to number of irq lines. Below screen shot shows the irq descriptor table.

This descriptor had two main pointers handler and action. Handler pointer is to managing a irq line like enabling , disabling and acknowledge and action pointer is to call the registered function to be called.

Linux allows single irq line to be shared between multiple devices in such case the action pointer points to multiple handler functions.Each ISR is called in sequence of the ISR's registered.



1. In a processes address space does the code segment starts from zero?

Ans. No, out of 0-3GB processes address space the first 64kb is not used for process address space

that 64KB is used by OS for critical/important tasks.

(In general it is 64kb it may vary.)

Nice link for LDD questions:

<http://embedded-telecom-interview.blogspot.in/2010/07/linux-and-linux-device-driver-interview.html>

```
Void main()
{
 char *str_ptr="MY_STRING";
 printf("%s",str_ptr);
}
```

In processes address space where is string "MY\_STRING" stored ?

Ans. string "MY\_STRING" stored at read only Text segment and str\_ptr stored at data segment.

**In a Process Address Space what if stack or heap collide?**

**Malloc Implementation (not Interview FAQ but you can impress interviewer if asked)**

<http://danluu.com/malloc-tutorial/>

(

Very effective and easy to understand (Myself read this down to the blog you can find very useful link <https://github.com/danluu/malloc-tutorial/blob/master/malloc.c> to see simple malloc code)

Read this too

<http://duartes.org/gustavo/blog/post/anatomy-of-a-program-in-memory/>

)

[http://www.inf.udc.cl/~leo/Malloc\\_tutorial.pdf](http://www.inf.udc.cl/~leo/Malloc_tutorial.pdf)

[https://fossies.org/dox/glibc-2.21/malloc\\_8c\\_source.html](https://fossies.org/dox/glibc-2.21/malloc_8c_source.html) //glibc malloc source code.

<http://moss.cs.iit.edu/cs351/assets/slides-malloc.pdf>

Ericsson Interview from WIPRO:

1. Write a simple with critical section, mutex lock and mutex unlock.
2. What is the difference between Mutex and Binary semaphore
3. What is difference between Mutex and semaphore
4. Working of spinlocks
5. Can Spinlocks work on multiprocessor system.
6. In what scenarios mutex is used and semaphore is used.
7. Delete a particular node in a linked list having argument passed as address of a node.  
without Header pointer known and other implementation with header pointer known
8. When a signal is raised on an interrupt line what will be passed to processor either interrupt number or anything else?
9. How kernel comes to know which device raised an interrupt when interrupt is shared.
10. How can an application talk to specific device in this below case?  
having Hard Disk devices
  - SATA0
  - SATA1
  - SATA2if there are drivers registered for these devices and a single driver is used to manage these devices then how can an application talk to a specific device  
Ex: if I want to read from SATA1 hard disk and Write to SATA0 and SATA1 hard disk.
11. How do you establish a sync mechanism in above hard disk scenarios.
12. When a Mutex lock is acquired by a low priority task and High priority tries to acquire the lock  
will the low priority task be pre-empted.
13. The same above scenario is with sema and spinlock
14. Explain working of any IPC mechanism.
15. Where your driver will be in kernel
16. Explain about process address space with data seg, code seg..  
how much memory is occupied by process address space.
17. When a same executable is executed in two terminals like terminal 1 executes ./a.out and terminal 2 executes ./a.out what will the program address space look like on RAM
18. If a global variable defined as int V1=100 in programme\_1 and modified this V1=200 in programme\_2. what will be printed in Programme\_1 and Programme\_2.
19. char \*name="WIPRO";  
    sizeof(\*name) = ? (Ans :1)  
    sizeof("WIPRO")= ? (Ans :6 )  
    sizeof(name)= ? (Ans :4 )
20. Declare a function pointer which takes 2 arguments as char pointers, and returns an integer pointer?
21. For above function pointer assign the function to be called?

CISCO Interview from WIPRO:

Malloc allocation tutorial

Write your own malloc implementation without using system calls.(FAQ Interview)

<http://danluu.com/malloc-tutorial/>

[http://www.inf.udec.cl/~leo/Malloc\\_tutorial.pdf](http://www.inf.udec.cl/~leo/Malloc_tutorial.pdf)

Understanding linux Kernel Oreily

<http://idak.gop.edu.tr/esmeray/UnderStandingKernel.pdf>

Linux Kernel Development by Robert Love click on 1st item

<https://www.google.co.in/webhp?sourceid=chrome-instant&ion=1&espv=2&ie=UTF-8#q=robert+love+linux+device+drivers+4th+edition+pdf>

Linux Device Drivers Oreilly

<http://www.oreilly.com/openbook/linuxdrive3/book/>

Below link explains i2c well

<http://maxembedded.com/2014/02/inter-integrated-circuits-i2c-basics/>

Below Link Explains how a Prob function called (FAQ in device driver interviews)

<http://stackoverflow.com/questions/7578582/who-calls-the-probe-of-driver>

1. Write program for Fibonacci series of first n number

Ex: if n=4 ,answer is 0,1,1,2

2. Write program to Insert element in static array.

Ex: int a[20] = {1,2,3,5},

Given input element is 4,

answer is {1,2,3,4,5}

3. Write your own malloc implementation without using system calls.

Hint: How you will get the heap next free address

4. Reverse the words in given strings.

Ex: I am good boy

answer : boy good am I

5. Swap two element in efficient manner

### Design Part

1. Class diagram for State Machine.

OS Concept

1.Message queues

2.semaphore

Memory Leak Debugging concept. Whats your approach on debugging a core dump.

Memory layout of a process. Stack/Heap/BSS/Data/Code.

implement malloc API.

C++ Programming Concepts . why do we need virtual function? How the performance is affected by this?)

Concepts of pointer and reference.

1. Write a malloc\_alloc() function for an integer, without using heap or any system calls

Int \* malloc\_alloc(int \* 1) Hint: normal integer pointer allocation ; storage can be in data segment

2. Write a pgm to insert an element in a sorted array; check all boundary conditions

3. Implement a state diagram for state change based on events

Hint: use a Table mapping events to state change

4. Memory usage, heap, physical memory, system concepts

5. Data segment/Bss etc

6. Compare the size of the 2 files after compilation.

File1:

```
int arr1[1000];
```

```
Main() {
```

```
}
```

File2:

```
Int arr2[2000];
```

```
Main() {
```

```
}
```

Hint: File size is same

7. Lastly worked project details and design

8. Bit manipulation – To set a bit at a position in a 32 bit integer

### Cisco f2f interview

1. Write a C program to set a bit at a particular position.
2. Write a function in library file that function should return a 32bit unique integer number every time the function is called and that function will not take any arguments?
3. How above written library function to be take care in multi thread scenarios.
4. What is the default value of static?
5. how the default value is get assigned to static variables?
6. Where static variable are present in programme address space?
7. How a interrupt is registered?
8. How a delayed execution in interrupt is handled?
9. In detail about top half and bottom Half?
10. In which interrupt context the interrupt will run?  
Ans. Arbitrary Thread context means(The processes in which the interrupt was triggered.)  
[\(https://www.osr.com/blog/2014/09/08/arbitrary-thread-context-article-video/\)](https://www.osr.com/blog/2014/09/08/arbitrary-thread-context-article-video/)
11. what all Debugging techniques you know?
12. do you know scripting?

### **2nd-Round:**

1. Create a pointer pointing to integer type point that to char array containing 10 elements?
2. The above created pointer if incremented ++ which element of the array it points to?
3. Write an instruction such that integer pointing sould point to first element of the array on increment?
4. Will their be any compilation errors while trying to print the below?

```
int *ptr=NULL;
printf("%d",*ptr);
```
5. will their be any warning if yes when at compile time or run time?
6. Expalin about your latest project?
7. Explain about shared interrupt and how an correponding  
interrupt Handler is called while an interrupt is shared?

### Broad com:

1. write a C programme to check given string is palindrome or not.
  2. Write a C programme to reverse a string.
  3. write a C programme to copy a string to other char pointer
  4. write a C programme for any sorting logic.
  5. write a C programme to set a particular bit to 1 for given position.
  6. write a C programme to set a particular bit to 0 for given position.
  7. write a C programme to toggle a particular bit for given position.
  8. Volatile keyword
  9. Interrupts mechanism
  10. Synchronization techniques(Semaphore/Mutex/Spinlocks)
- 

### SmartPlay :

1. write a C programme to set bits from 3 to 7 positions to 1 in a 32-bit given value  
Condition is at a time only one bit from 3 to 7 bit positions to be 1 and if not all bits to be reset.?

2. Determine the minimum stack size required for given prog by end of main function?

```
Void main()
{
int x,y;
char c;
fun1();//Assuming forward declaration is done.
fun2();//Assuming forward declaration is done
fun3();//Assuming forward declaration is done
}

void fun1()
{
int a,b,c;
c=a+b;
}

void fun2()
{
char c;
}

void fun3()
```

3. what is the internal implementation/mechanism made to work volatile to get the updated value?

4. Behavior of system call from user level to kernel level.

5. What is software interrupt and hardware interrupt with examples?

### 6. Spin locks behavior?

7.What is structure padding?How can we eliminate it?

8.If structure padding is efficient why not C language made this as default behaviour?

9.

---

HCL telephonic:

1.Explain camera sensor project

2.How frame buffers are allocated in camera sensor project

3.how to debug your driver if u had any buffer issues

4.Do u know about V4L2 layer

5.Have u faced any page faults in driver and how u will solve that issue?

6.How the data is transferred from camera sensor to LCD display

7.Do u know about IOCTL operations? and have they been used in your camera sensor project.

8.what is softirq and tasklet and work queues?

9.how you select the bottom half mechanism and what are the differences between them

10.what in interrupt context and process context

11.what is context switch

12.what are kernel synchronization techniques

13.Difference between sempahore and mutex and when u select the semaphore and mutex to use in your driver

14.what is process management

15.What is kernel thread?

16.what is pthread?

17.Do u know about IPC mechnanisms?

18.what is shared memory

19.Explain MMU in Linux

20.what is /proc entry and how it is useful

21.how the device files are created in linux

22.Difference between structure and union

23.what is function pointer and how it is used?

24.can i call a function using the address of that function through a point variable?

---

General Question Collection:

->Storage classes

->Where static and global data members stored?

->What is shared lib and static library?

->How can two slaves can communicate with master in i2c protocol at a time?

->How can we address different devices from same vendor i2c?

->Structure padding?

->Difference between #define and CONST, which one will you chose in programming?

->Debugging techniques?

- >Volatile keyword?
- >Top half and bottom half interrupt approach?
- >Inter process communication techniques?
- >Threads, Fork and clone?

- >Device driver projects
  - >write a efficient programme to set a particular bit in a givne number?
  - >Difference between typedef and #define?
  - >Callback functions?
  - >Intterupt top half and bottom half?
  - >USb driver project?
- 

...

Global edge:

- 1.Explain camera sensor driver project
  - 2.where the camera driver resided in kernel
  - 3.Know about VL42
  - 4.I2C protocl
  - 5.I2C- dummy write,repeated start,arbitration,synchronization between masters
  - 6.Lm475 temeprature driver project
  - 7.storage classes in C
  - 8.Questions on static,diff b/n static and extern
  - 9.can we assign pointer to a static variable?
  - 10.can we assign pointer to a register variable
- 

...

HCL:

- 1.How to give memory of 250Mb to kernel (high zone memory)
  - 2.Working of camera sensor project
  - 3.bit operations :set a bit, clear a bit,toggle a bit
  - 4.find a loop in a linked list
  - 5.can we add two pointers
  - 6.can we subtract pointers
  - 7.Difference between structures and union
  - 8.storage classes
  - 9.diff between static and extern
  - 10.How to develop a device driver for a hard device?
- 

...

DELL:

- 1.Explain the projects what u have done
  - 2.IPC,memory management, process management
  - 3.Socket programming
  - 4.Delete a node in the list
  - 5.find a word in a buffer
- Telephonic for CISCO:

Linux

1. Explain in brief about your project (Camera device driver)?
2. What is the communication protocol used to communicate with camera sensor?
3. How to write data to camera sensor address?
4. Which type of address (virtual or physical) is used in function iowrite32?
5. What is concept of virtual address or physical address?
6. Why virtual address is required when physical address is existing?
7. What all the things to be taken care while handling an interrupt?
8. Main difference between Tasklets and workqs?
9. When an interrupt can be enabled again? (Not asked but better to find the answer ◊ When an interrupt can be disabled)
10. What is volatile variable?
11. How to register an interrupt handler?
12. How to know whether a interrupt is an edge triggered or level triggered from registering an interrupt handler?
13. What is the type of driver char or n/w or block driver you developed?
14. How to register a CHAR driver to kernel?
15. How to load a dynamic driver or static driver?

C

11. what is a static variable or static function?
12. what is the difference between static variable and global variable?
13. Where a global variable and static variable resides in at run time?
14. What all the segments present when a programme at compile time?
14. What all the segments present when a programme at run time?
15. Tell me the logic to disable specific set of bit provided a 32 bit number, start bit and end bit?  
Example given numb, disable whatever the data from 5th bit to 10th bit.
16. Tell me the logic how can you find depth of a stack?  
Example In a program their is foo1() function calling foo2() function calling foo3() in any function write a logic to know the depth of stack.
17. s3c2440 processor supports which ENDIAN type?
18. Can Volatile be applied to pointer?
19. Why to have Virtual memory concept instead why cannot we increase the RAM on system?
- 20.s3c2440 supports what bit type 32/16/8?

symphony teleca F2F questions

- 1.volatle variable usage
- 2.const and volatile difference
- 3.can volatile and const be used together?
- 4.How do you declare pointer to memory mapped area?
- 5 Call backs and explain with example
- 6.How callbacks are different to normal function calling?
- 6.synchronization mechanisms (semaphore and mutex)
- 7.Difference between binary semaphore and Mutexes
- 8.Interrupt handler ? how interrupt is serviced?
- 9.difference between Malloc and calloc
- 10.what is memory leak and how to avoid it?
- 11.I2C protocol

Second round:

- 1.What are the design issues/considerations that was taken care for your camera project.
  - 2.Explain about the S3C2440 processor architecture  
allegion interview F2F qs
  - 1.why size of data types is restricted?
  - 2.why volatile and why it cant be used to all variables
  - 3.difference b/w macro and a constant variable
  - 4.I2c protocol and diff b/w i2c and spi
  - 5.what is i2c bus error
  - 6.how to synchronize devices which operates at different frequencies
  - 7.explain how an interrupt of hardware device is handled
  - 8.explain embedded system architecture (von neumann and harvard architecture)
  - 9.s3c2440 board architecture
  - 10.make file creation and its rules and how it works to create .objs
  - 11.Build process of C program (preprocessing, compilation, linking etc)
  - 12.Questions on Wifi-- related to networking
- 

...  
Broadcom//Face to face.

1. WAP to toggle the given range of bits for a given number(without using loops)?
  - 2.How C programme memory address space is allocated?  
(stack,heap ,data segment,code segment)  
given few functions in a programme and asked to tell which all variables goes to which sections in programme memory address space)
  - 3.How can you make a header file not to include multiple times?
  - 4.Q's on #ifdef,#ifndef
  - 5.where are static variable get stored in programme address space?(in stack/heap/data segment)
  - 6.if we print the address of local variable which address it would print?
  - 7.What all the data members stored on data segment?
  - 8.Volatile use case?volatile is qualifier or not?
  - 9.static use case ?
  - 10.Over all more Q's on programme address space,stack heap segments?
- 

...  
Broadcom//2nd Round

- 1.main()

```
{\nint a;\n}
```

where a will be stored.
- 2.WAP to know how stack grows in upward or downward directions?
- 3.WAP to convert ASCII to Integer?
- 4.Why ASCII number is required?
- 5.Given Two boards and with exactly same CPU's should they require different compilers?
- 6.#include<stdio.h>

```
void main (){\nint a;
```

}

how this programme looks after preprocessing?

8.The project done by you are in linux platform if gcc is the compiler used what is the preprocessor used?

9.what is the job of preprocessor?

10.more deep question on preprocessor and compiler?

11.what is diff b/w process and threads?

12.Will threads have their own stack space?

13.can one thread access the address space of another thread?

//<http://www.cs.cf.ac.uk/Dave/C/node29.html>

14.Overall 2nd round more Q's on preprocessor,compliler,process and threads?

15.main()

{

int a;

}

how variable 'a' get allocated memory in assembly level language?

16.In printf what it the purpose of %X ?

17.What is the size of integer variable on 32bit and 64bit machine?

18.Structure

{

Int a,

Char c;

Int b;

}

What is the size of the structure.

19. What is an re-entrant function?

---

....

Broadcom://Telephonic

1. Brief about yourself?

2. In Depth of USB Device Driver?

a. What are the different data transfer modes?

b. If there is a speech audio data to be received by USB device what type of data transfers will you chose.

c. What are the different types of device descriptors?

d. How many END point descriptors can be present per device?

e. What is a configuration descriptor?

f. What is a control type of data transfer?

3.In a given structure having n number of structure members how do you find the sizeof structure?

4. How can you find the offset of next structure member

if out of n-number of structure members ,xth structure member is given how can you find the size of structure?

5. DO you know container macros? In all most all kernel drivers it is used?

6. What is call by value and call by reference?

7. If passing name is an call by value, then array name is passed is it also call by value?

8. Volatile keyword? Usage?

9. Can a volatile member can be cached?

10. Can you see any difference between MUTEX and Binary Semaphore?
  11. Spinlocks mechanism?
  12. A global data member is programmed to get modified in a thread and Interrupt, so what kind of synchronization mechanism will you use?
  13. If spin locks what flavor of spin locks will you use.
- 

---

Xilinx:

How can a static driver runs? Without doing any insmod?

Any idea about Device tree?

synchronization Techniques programmes

IOCTL implementation prog

C-Code data segments

C-programme compilation steps

Where will be your driver file will be located

What is the entry point of your driver

Memory Kernel Working

USB Driver

i2c Driver

IPC Mechanisms

Port Mapping/Memory mapping

Fork/Clone

Diff of Kmalloc and Vmalloc

Linked List

Device Tree

String operations

Call back

Programme to find little Endian and Big endian

programme to find the size of a structure with size of operator.

Programme to implement reversal of bits

ARM processor overall knowledge.

Overall knowledge of kernel building knowledge.

---

---

MiroChip

-> Linked List creation, adding elements, insertion in the middle, deleting elements in the middle.

-> 2-Dimensional array creation, allocation

-> Reversing bits.

-> IPC

-> Creating of threads

-> Difference between process and threads

-> Debugging of kernel

-> Kernel crash dump analysis

-> basics of board support package

---

---

Tata Elexsi:

- >Storage classes
  - >Where static and global data members stored?
  - >What is shared lib and static library?
  - >How can two slaves communicate with master in i2c protocol at a time?
  - >How can we address different devices from same vendor i2c?
  - >Structure padding?
  - >Difference between #define and CONST, which one will you chose in programming?
  - >Debugging techniques?
  - >Volatile keyword?
  - >Top half and bottom half interrupt approach?
  - >Inter process communication techniques?
  - >Threads, Fork and clone?
  - >
- 
- ...

### Global Logic:

- >Device driver projects
  - >write a efficient programme to set a particular bit in a given number?
  - >Difference between typedef and #define?
  - >Callback functions?
  - >Interrupt top half and bottom half?
  - >USB driver project?
  - >
- 
- ...

### General:

- ◊Sizeof operator without using sizeof operator?
- ◊Search the middle element in a given list?
- ◊Ethernet driver how a packet get transferred in Ethernet drivers?

- ◊System call flow?
- ◊blocking and non-blocking USB calls
- ◊How and steps a C programme executes.,
- ◊How the programme is loaded in execution space
- ◊

- 1.C prog Compilation steps
- 2.C prog memory layout.
- 3.Difference b/w string and string lit
- 4.What actually preprocessor does

5.

---

---

WIPRO:

- >How Memory Management works in KERNEL?(In total about Kernel Memory management)
  - >Inter Process communication techniques?
  - >Spin locks working?
  - >Favorite topics in kernel?
  - >Tell about interrupts handling in kernel?
  - >When a interrupt is handled whether that specific interrupt is disabled or not? If disabled why ? if not disabled why not disabled?
  - >When a interrupt is handled tell whether all interrupts are disabled or not ?if yes why ?if no why?
  - >As device driver developer when you will you disable or enable particular interrupt or all interrupts>
  - >What is the functionality of save\_interrupts?
  - >what is the first function called when interrupt is handled?
  - >Have you heard about "Handle\_interrupt" ?
  - >What are the possible ways that memory leaks can happen in a programme ?(Apart from allocation and not freeing )
  - > How a memory leak can happen in a linked list?
  - >How a Reception of packet performed in a Ethernet device driver? (Explain from starting to reception of packet)
  - >How can you rate yourself in KERNEL?
- 
- 

HARMAN

- >How can you avoid accessing of an array beyond its limits?
  - >Whether an exception or normal operation when an array is accessed beyond its limits?
- 
- 

General Q's:

- >What are types of function(Callback,Recursion ....)?
  - >what is advantage and disadvantages of each function type?
  - >Implementation example for each function type?
  - >What are different searching algos explain each one ,implement any one?
  - >link for programmes of palindrome ,fibonacci,searching etc.  
<http://study-for-exam.blogspot.in/search/label/C%20programming#.VHUuByWoLFY>
- 
-

WIPRO:

- >How Memory Management works in KERNEL?(In total about Kernel Memory management)
  - >Inter Process communication techniques?
  - >Spin locks working?
  - >Favorite topics in kernel?
  - >Tell about interrupts handling in kernel?
  - >When a interrupt is handled whether that specific interrupt is disabled or not? If disabled why ? if not disabled why not disabled?
  - >When a interrupt is handled tell whether all interrupts are disabled or not ?if yes why ?if no why?
  - >As device driver developer when you will you disable or enable particular interrupt or all interrupts?
  - >What is the functionality of save\_interrupts?
  - >what is the first function called when interrupt is handled?
  - >Have you heard about "Handle\_interrupt" ?
  - >What is Memory Leak?
  - >What are the possible ways that memory leaks can happen in a programme ?(Apart from allocation and not freeing )
  - > How a memory leak can happen in a linked list?
  - >How a Reception of packet performed in a Ethernet device driver? (Explain from starting to reception of packet)
  - >How can you rate yourself in KERNEL?
  - >Points you will inquiry when a driver from one platform to be ported to another platform?
  - >How do you do debug?
  - >Have analyzed kernel panic code dump(what all data it dumps and what do you understand from it)
  - >What are storage classes available ?what are the scenarios the specific storage class is used?
  - >Are you comfortable with pointers, linked list and Queues?
  - >What is memory mapping ?How kernel knows to which memory location the devices are mapped?
  - >
- 
- ....

HARMAN

- >How can you avoid accessing of an array beyond its limits?
  - >Whether an exception or normal operation when an array is accessed beyond its limits?
- 

....  
Aricent :

- 1.What is Function Pointer?
- 2.What is Call back function?
- 3.How Call Function works?
- 4.What is the clock frequency used in your I2C driver designed?
- 5.What is the start bit condition in I2C?

- 6.How I2C protocol works?
  - 7.What are different speeds the I2C had?
  - 8.What is the path of your driver inside kernel?
  - 9.Can interrupts sleep?why?
  - 10.Types of interrupts?
  - 11.What are the function names for the driver you designed?
  - 12.How the master knows what is the start condition ?
  - 13.while in between I2C communication what happens if clocks happens to be dragged low which is not as per i2c standard?
  - 14.What if the slave device is not responding or no acknowledge bit is sent by slave device?
  - 15.What is the difference between kmalloc and vmalloc?
  - 16.how to get physically contiguousness memory allocation if kmalloc is giving logical contiguousness allocations?
  - 17.How do a driver get registered to kernel?
  - 18.What actually happens while interrupt occurs exactly explain in terms of context switching ,what happens to the current running process?
  - 19.As their no code written to do context saving when a interrupt happens in any of the functions then which is responsible in doing this context saving and switching context?
  - 20.How "Make" is performed in compiling a module?
  - 21.What is VOLATILE keyword?
  - 22.Write a program to perform a string copy without using string library functions?
- 
- ...

### Wipro/ERRICSON:

- 1.What is the difference between Linux and RTOS ?and Reason why linux is not Real time?
  - 2.How User space buffer can be accessed from kernel level and viceversa?
  - 3.How to debug different threads in GDB?
  - 4.Comfort level in Assembly level language?
  - 5.What are the kernel structures and which kernel structure used in device driver development?
  - 6.What is interrupt nesting, How IRQ priority handled?
  - 7.What is SERDES?
  - 8.How do you pass data between kernel modules?
  - 9.What are different embedded design patters?
  10. Memmory Mapping, synchronization Techiniques , Interrupts , Mutes , Interprocess\_communication?
- 
- ...

### Ericsson :

- 1.Explain about any Device Driver project your good at?
- 2.Can call back functions or Recursive call can be made in asynchronous events?
- 3.What is stack overflow?
- 4.Difference between processes and Threads?
5. How can you corrupt STACK?
- 6.What are the issues faced while developing driver projects?
- 7.What are your strengths?
- 8.Rate your self in C?

9.What is Blocking and Pooling?Diff ?

---

---

UTC Aerospace:

- 1.Storage classes in C
  - 2.What is pointer
  - 3.What is volatile
  - 4.What is Structure
  - 5.What is Class in C++
  - 6.Difference between structure and class
  - 7.What is Partition on DO-178B document?
  - 8.What is Real time system
  - 9.What are types of real time systems
  - 10.what happens in Real time system if specific task is not executed at specified time?
  - 11.What is Dead code and Deactivated code and diff b/w them?
  - 12.Level of software as per DO-178B?
- 

...

Aricent:

- 1.Explain about USB device Driver?
  - 2.Explain about I2C Driver?
  - 3.How USB driver is registration mechanism?
  - 4.How to pass a two dimensional array to other function?
  - 5.Syntax of main with arguments?
- 
- 

Global Edge:

C:

- 1.Difference between string and string literal
- 2.compilation steps of a C programme
- 3.Memory allocation of a c programme
- 4.Why uninitialized data is placed in uninitialized data segment?

Device drivers project:

- 1.More questions on Camera device driver project?and more Questions specific to project?
- 

...

SMART\_PLAY

- 1.How do find whether a processor is Big Endian/Little Endian?
  - 2.Intreputs,Bottom half.
  - 3.Memory allocation using Kmalloc with FLAGS?
- 
- 

...

synapse-da

Device Drivers

Camera Project

How do you get JPEG image from CAM

Diff Semaphore & Mutex

Diff Insmod and Modprobe

Diff Binary sema and Muex

What are the precautions to be take care in ISR routine.

Diff Process and Thread

How to create a Major and Minor Numbs dynamically.

Diff b/w char ,Block and Network drivers

When Kernal had a previlage to write to any memory location then why do we need copy\_to\_user and copy\_from\_user functions.

What is Dead lock scenario.

Diff Kmalloc and Vmalloc

---

.....  
....

C:

Volatile

Diff array and Linked List.

Storage classes and difference.

Usage of Static data members.

While and For loop efficiency and which is better.

How to find a number is divisible by 4 without using mathematical operators

Diff static library dynamic library and shared library

Diff inline and macro.

Diff malloc and calloc

...

HARMAN:

- 1.How to create buffers in Kernel?
- 2.What is the difference b/w kmalloc and vmalloc
- 3.How buffer is copied to user level
- 4.Explain working of interrupts
- 5.Explain about the projects mentioned in CV?
- 6.How to search for a specific nth-node in a given unknown number of nodes in a linked list?
- 7.How to determine the size of structure without using "sizeof" operator?
- 8.What is memory leak
- 9.Function Signature of registering Interrupt handler? and Flags used while requesting IRQ?
- 10.Explain Shared memory ,Message Queue Synchronization techniques in Linux

...

Unknown Consultancy:

Volatile variable:with example

Volatile is qualifier applied while declaring of a variable.

Volatile keyword tells the compiler that variable can be modified from outside source without the action of the code the compiler finds nearby.

Volatile int a,

Int volatile b,

Int volatile \*foo-> Pointer pointing to a volatile integer

Volatile int \*foo;

Register variable

Unions with practical example

Malloc and its disadvantages

Int \*const p and const \*int p meaning

Endianness determination

Diff between array and linkedlist

Bitoperations : how do u set a bit?

---

Metasoft--Graphite semi

- 1.What happens when insomd is performed?
  - 2.How system call is executed?
  - 3.What all steps involved while a C programme .exe is created?
  - 4.How can you make part of driver code is allowed to be executed and not executed depending upon condition?
  - 5.How Interrupts are executed?.
  - 6.Different synchronization techniques ?how do you chose the best sync mechanism fits the scenario?
  - 7.What are all the Kernel APIs used in your driver programming?
  - 8.In a given list of elements how to find the middle element?
    - condition no count is available
    - condition not known how many elements present
  - 9.How the data integrity check is done in case of network packets or I2c driver communication?
  - 10.How a packet is sent in ethernet driver?
- 

Consultancy:

1. A = 10000000000

B = 0110

i = 2

j= 6

o/p =10000001100??

2.how to implement own malloc function.

3.how to delete duplicate node in single linked list.

4.what is the entry points in kernel?

5.how to create a major number dynamically?

6. ISR descriptor Table

7. spin\_lock\_bh() diff spin\_lock\_irq()?

8. Do\_irq?

9. Static Drivers vs Dynamic Driver?

10. Diff SLAB and Vmalloc

11. copy\_to\_user and copy\_from\_user?

12. how to divided critical work and non critical work based on top half and bottom Half

13. int arr[] {9,9,9}

o/p = {1,1,1,0}

.....

Global logic interview questions

What is system call? How to check what are the system calls used?

What is linux kernel module programming?

What is boot loader?

What ioctl?

How to write own module?

Programming questions:

Printing \* in triangle shape , input is number lines to print

```
*
* * *
* * * * *
```

Delete a node in the linked list and the node to be deleted is the one that is passed to the function, no head reference is given

Print nth node from the end in a single linked list

String is recursive or not, s1 = "abcd" , s2 is the user input – s2 can be cdab, dbca

SONY india questions

1. How to speed up the memcpy, have you seen memcpy code in kernel ?
2. How to build the vmlinux image
3. How to debug crash, what is the first line you see when you see a crash/oops message, explain crash console
4. How Ethernet device is registered
5. Have you checked linux lib code
6. How to debug kernel code
7. What is x86 and what are the registers used
8. How to write a makefile to build a customized kernel
9. More questions on debug and crash point related
10. Struct test { int a; int b; }; Assume you don't know the contents of structure, and you have the base address of test strucuture alone, how to do access the data inside it ??
11. How to register, allocate NIC.
12. How does system call works, in detail.
13. ARM or x86 archiecture
14. Step by step of writing a device driver structure, lot of cross questions, not a straight forward question
15. /proc file system, how to create it
16. ARM
17. In-depth of the projects worked so far... ↴

Few more Questions @

<http://linuxinterviewpreperation.blogspot.in/2013/01/linux-kernel-and-device-drivers.html>

---

.....

### References:

1. Steps involved in creating a C programme executable  
<http://www.thegeekstuff.com/2011/10/c-program-to-an-executable/>

2.Re-entrant functions and Thread safe is explained well at :  
<http://www.thegeekstuff.com/2012/07/c-thread-safe-and-reentrant/>

---

.....

## c interview programs for freshers

### 1. C programto print hello world without using semicolon

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

```
int main()
{
 if (printf("hello world"))
 {
 }

}
```

---

## 2. C program to find even or odd

```
#include<stdio.h>
int main()
{
 int n;
 printf("enter the number\n");
 scanf("%d",&n);
 if(n%2==0)
 {
 printf("even number\n");
 }
 else
 {
 printf("odd number\n");
 }
}
```

---

## 3. C program to find even or odd using bitwise operator

```
#include<stdio.h>
int main()
{
```

```
int n;
printf("enter the number\n");
scanf("%d",&n);
if(n&0x01)
printf("odd number\n");
else
printf("even number\n");
}
```

---

**4. C program to find even or odd using goto**

```
#include<stdio.h>
int main()
{
int n;
printf("enter the number\n");
scanf("%d",&n);
if(n%2==0)
 goto even;
else
 goto odd;
even: printf("even number\n");
 return;
odd: printf("odd number\n");
 return;
}
```

---

**5. C program to find greatest among three numbers**

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

```
int main()
{
 int a,b,c;
 printf("enter the three numbers\n");
 scanf("%d%d%d",&a,&b,&c);
 if(a>b && a>c)
 {
 printf("%d is greatest",a);
 }
 else
 {
 if(b>c && b>a)
 {
 printf("%d is greatest",b);
 }
 else
 {
 printf("%d is greatest",c);
 }
 }
}
```

---

#### 6. C program to find palindrome number or not

```
#include<stdio.h>
int main()
{
 int n,rem,rev=0,temp;
 printf("enter the number\n");
 scanf("%d",&n);
```

```
temp=n;
while(temp!='\0')
{
 rev=rev*10;
 rev=rev+temp%10;
 temp=temp/10;
}
printf("reverse number:%d\n",rev);
if(rev==n)
 printf("%d is palindrome number\n",n);
else
 printf("not palindrome number");
}
```

---

## 7. C program to generate fibanacci series

```
#include<stdio.h>
int main()
{
 int a=0,b=1,n,i,c;
 printf("enter the number\n");
 scanf("%d",&n);
 for(i=0;i<n;i++)
 {
 c=a+b;
 a=b;
 b=c;
 printf("%d ",c);
 }
}
```

---

**8. C program to swap two numbers**

```
#include<stdio.h>
int main()
{
 int a,b,temp;
 printf("eneter the two numbers\n");
 scanf("%d%d",&a,&b);
 printf("before swapping\n");
 printf("a=%d b=%d\n",a,b);
 temp=a;
 a=b;
 b=temp;
 printf("after swapping\n");
 printf("a=%d b=%d",a,b);
}
```

---

**9. C program to swap two numbers without third variable**

```
#include<stdio.h>
int main()
{
 int a,b;
 printf("enter the two numbers\n");
 scanf("%d%d",&a,&b);
 printf("before the swapping\n");
 printf("a=%d b=%d\n",a,b);
 a=a+b;
 b=a-b;
 a=a-b;
```

```
a=a-b;
printf("after swapping\n");
printf("a=%d b=%d",a,b);
}
```

---

**10. C program to swap two numbers with bitwise operator**

```
#include<stdio.h>
int main()
{
 int a,b;
 printf("enter the two numbers\n");
 scanf("%d%d",&a,&b);
 printf("before swapping\n");
 printf("a=%d b=%d\n",a,b);
 a=a^b;
 b=a^b;
 a=a^b;
 printf("after swapping\n");
 printf("a=%d b=%d",a,b);
}
```

---

**11. C program to find string palindrome or not**

```
#include<string.h>
#include<stdio.h>
int main(){
 char str[20],rev[20];
 int i,j;
 printf("\nEnter a string:");
```

```
gets(str);
for(i=strlen(str)-1,j=0;i>=0;i--,j++)
 rev[j]=str[i];
 rev[j]='\0';
printf("reverse string:%s\n",rev);
if(strcmp(rev,str)==0)
 printf("The string is a palindrome\n");
else
 printf("The string is a not palindrome");
}
```

---

**12. C program to find factorial**

```
#include<stdio.h>
int main()
{
 int n,fact=1;
 printf("enter the number\n");
 scanf("%d",&n);
 while(n>0)
 {
 fact=fact*n;
 n--;
 }
 printf("factorial %d",fact);
}
```

---

**13. C program to find prime number or not**

```
#include<stdio.h>
int main()
{
 int n,i;
 printf("enter the number\n");
 scanf("%d",&n);
 for(i=2;i<=n-1;i++)
 {
 if(n%2==0)
 {
 printf("not prime number\n");
 break;
 }
 }
 if(i == n)
 printf("prime number\n");
}
```

---

#### 14. C program to swap two nibbles

```
#include<stdio.h>
//void displaybit(int x);
int main()
{
 int n,temp1,temp2,result;
 printf("enter the number\n");
 scanf("%x",&n);
 temp1=n&0x0f;
 temp1=temp1<<4;
 temp2=n&0xf0;
```

```
temp2=temp2>>4;
result=temp1|temp2;
printf("%x",result);

}
```

---

### 15. C program to find Armstrong number

```
#include<stdio.h>

int main()
{
 int sum=0,n,temp,d,rem;
 printf("enter the number\n");
 scanf("%d",n);
 temp=n;
 while(n>0)
 {
 rem=temp%10;
 d=rem*rem*rem;
 sum=sum+d;
 temp=temp/10;
 }
 printf("%d",sum);
 if(n==sum)
 printf("armstrong number\n");
 else
 printf("not armstrong");
}
```

---

### 16. C program to sort numbers in ascending order

```
#include<stdio.h>
int main()
{
int a[5],i,j,t;
printf("Enter 5 nos.\n\n");
for (i=0;i<5;i++)
scanf("%d",&a[i]);
for (i=0;i<5-1;i++)
{
for(j=i+1;j<5;j++)
{
if(a[i]>a[j])
{
t=a[i];
a[i]=a[j];
a[j]=t;
}
}
}
printf("Ascending Order is:");
for(j=0;j<5;j++)
printf("\n%d",a[j]);
}
```

---

**17. C program to sort numbers in descending order**

```
#include<stdio.h>
int main()
{
```

```
int a[5],i,j,t;
printf("Enter 5 nos.\n\n");
for (i=0;i<5;i++)
scanf("%d",&a[i]);
for (i=0;i<5;i++)
{
 for(j=i+1;j<5;j++)
 {
 if(a[i]<a[j])
 {
 t=a[i];
 a[i]=a[j];
 a[j]=t;
 }
 }
}
printf("Descending Order is:");
for(j=0;j<5;j++)
printf("\n%d",a[j]);
}
```

---

**18. C program to print ASCII values from 0-255**

```
#include<stdio.h>
int main()
{
 int i,a=0;
 for(i=0;i<=255;i++)
 {
 printf("%d:%c\n",a,i);
 }
}
```

```
a++;
}

}
```

---

**19. C program to convert decimal to binary number**

```
#include<stdio.h>

int main()
{
 int n,i,mask;
 printf("enter the number\n");
 scanf("%d",&n);
 for(i=31;i>=0;i--)
 {
 mask=0x01<<i;
 if((mask&n)==0)
 {
 printf("0");
 }
 else
 {
 printf("1");
 }
 if(i%4==0)
 printf(" ");
 }
}
```

---

**20. C program to perform implicit conversion**

```
#include<stdio.h>

int main()
{
 char a='A';//implicit conversion the value of A is converted in to as a ASCII value
 int b=5,c;
 c=a+b;
 printf("%d",c);
}
```

---

**21) C program to find even or odd?**

```
#include<stdio.h>

int main()
{
 int n,z=0x01;
 printf("enter the number\n");
 scanf("%d",&n);
 (n&z) && printf("odd")||printf("even");

}
```

---

**22) C program to display the alphabets in given range?**

```
#include<stdio.h>

int main()
{
 int n,m,i;
 char a[]={ 'A','B','C','D','E','F','G','H','I','J','K','L','M','N','O','P','Q','R','S','T','U','V','W',
 'X','Y','Z'};


```

```
printf("enter the range of numbers\n");
scanf("%d%d",&n,&m);
for(n;n<=m;n++)
{
 printf("%c",a[n-1]);
}
}
```

---

23) C program to find given number even or odd without using if condition?

```
#include<stdio.h>
int main()
{
 int n,z=0x01;
 printf("enter the number\n");
 scanf("%d",&n);
 (n&z) && printf("odd")||printf("even");

}
```

---

24) C program to perform explicit type conversion?

```
#include<stdio.h>
int main()
{
 int a=5,b=3;
 float c,d;
 c=a/b;
 printf("%d\n",c);
 printf("%f\n",c);
 d=(float)a/b; //explicit conversion
}
```

```
printf("%f",d);
}

```

**25) C program to find given year leap year or not?**

```
#include<stdio.h>
int main()
{
 int n;
 printf("enter the year\n");
 scanf("%d",&n);
 if(n%4==0 || n%100==0)
 {
 printf("leap year\n");
 }
 else
 {
 printf("not leap year\n");
 }
}
```

---

**26) C program to print multiplication table?**

```
#include<stdio.h>
int main()
{
 int table,n=1,mul,i;
 printf("enter the number\n");
 scanf("%d",&table);
```

```
for(i=0;i<10;i++)
{
 mul= table*n;
 printf("%d*%d=%d\n",table,n,mul);
 n++;
}

```

**27) C program to count no.of digits in given number?**

```
#include<stdio.h>
int main()
{
 int n,c=0;
 printf("enter the number\n");
 scanf("%d",&n);
 while(n>0)
 {
 n=n/10;
 c++;
 }
 printf("digits:%d",c);
}
```

---

**28) C program to perform operation by giving operator?**

```
#include<stdio.h>
int main()
{
```

```
int d;
char c;
int a=50,b=3;
// printf("enter the operator\n");
while(1)
{
printf("enter the operator\n");
scanf("%c",&c);
if(c==42)
{
d=a*b;
printf("product:%d\n",d);
break;
}
else if(c==43)
{
d=a+b;
printf("sum:%d\n",d);
break;
}
else if(c==47)
{
d=a/b;
printf("div:%d\n",d);
break;
}
else if(c==45)
{
d=a-b;
printf("sub:%d\n",d);
break;
}
```

```
 }
 else if(c==37)
 {
 d=a%b;
 printf("module:%d",d);
 break;
 }
}
```

---

29) C program to find product of two numbers without using \* operator?

```
#include<stdio.h>
int main()
{
 int result=0,i;
 int a,b;
 printf("enter the two nums\n");
 scanf("%d%d",&a,&b);
 for(i=0;i<b;i++)
 result=result+a;
 printf("product:%d",result);

}
```

---

30) C program to count string length?

```
#include<stdio.h>
#define max 50
int main()
```

```
{

 int i=0,n;
 char s[max];
 printf("entr the string\n");
 gets(s);
 while(s[i]!='\0')
 {
 i++;
 }
 printf("the length of the string is %d charecters",i++);
}
```

---

**31) C program to find sum and product of given number?**

```
#include<stdio.h>
int main()
{
 int n,sum=0,rem,product=1;
 printf("enter the number\n");
 scanf("%d",&n);
 while(n!='\0')
 {
 rem=n%10;
 sum=sum+rem;
 product=product*rem;
 n=n/10;
 }
 printf("the sum of given number: %d\n",sum);
 printf("the product of given number: %d",product);
```

{}

## C Programming Puzzles and Tricky Questions

### C Puzzles and Tricky Questions

|           |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
|-----------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| C Puzzles | <a href="#">C program to print hello world without using semicolon</a><br><a href="#">C program to round off floating point number to nearest integer</a><br><a href="#">C program to find execution time of a program in seconds</a>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
|           | <a href="#">C program to print numbers from 1 to N without using semicolon</a><br><a href="#">C program to implement your own itoa function</a><br><a href="#">C program to count digits of a number using logarithms</a><br><a href="#">C program to check if a given number is a power of 2 in single statement</a><br><a href="#">C program to print a 2D matrix row wise without using curly braces</a><br><a href="#">C program to convert any number to string in one line</a><br><a href="#">How to create your own header file in C</a><br><a href="#">C program to multiply a number with 7 using bitwise operator</a><br><a href="#">C program to reverse the digits of a number in just three statements</a><br><a href="#">C program to swap two variable using XOR bitwise operator</a>                                                                                                                                                                                            |
|           | <a href="#">C program to sum the digits of a number in single statement</a><br><a href="#">C program to check whether a number is in a range [min, max]</a><br><a href="#">C program to print a long variable using putchar function only</a><br><a href="#">Program to count trailing zeros of a number using bitwise operator</a><br><a href="#">C program to check whether a number is odd or even without using if else statement</a><br><a href="#">How to take a multiline paragraph as input from user</a><br><a href="#">C program to print memory representation of a C variables</a><br><a href="#">C program to implement your own sizeof operator</a><br><a href="#">C program to read a password string from user</a><br><a href="#">C program to print digits of a number in words without using if-else and switch case</a><br><a href="#">C Program to print numbers from 1 to N without using loops</a><br><a href="#">C program to return multiple values from a function</a> |

### Error checking questions on c

1. In order to assign attributes to the pointer itself, rather than to the pointed-to object, you put the attributes after the asterisk.

like ' char \*const p = &c; ' – True/False

**2. What is the output of the below code ?**

```
char **p = "Hello";
printf("%s", **p);
return 0;
```

**3. There is a char \* pointer that points to some ints, and what should be done to step over it ?**

**4. What changes has to be done to get the ouput as “9 8 7 6 5” by the same recursive call method ?**

```
int main(void) {
 int i=9;
 printf("%d \t",i--);
 if(i)
 main();
 return 0;
}
```

**5. What is the output in the below program ?**

```
void main() {
 int z = 12;
 printf("%d",printf("Mr.123\\\"));
 printf("%d",printf("%d%d%d",z,z,z));
}
```

**6. You can't 'shift' an array -**

True/False?

**7. Is it possible to do like this in C ?**

```
char s[8];
s="ABCD1234";
```

**8. bit-fields do not provide a way to select bits by index value – True/False?**

**9. Does 'char \*a' allocate some memory for the pointer to point to ?**

**10. A null pointer is does not to point to any object or function – True/False?**

An uninitialized pointer might point anywhere – True/False?

**11. The address of operator will never yield a null pointer – True/False?**

malloc returns a null pointer when it fails – True/False?

**12. char y[] is not identical to char \*y – True / False .?**

Explain the reason for either true/false.

**13. What would be output of the below code ?**

```
char x[] = "abcde";
```

```
char *y= "abcde";
```

```
printf("%c \n", x[3]);
```

```
printf("%c \n", y[3]);
```

**14. Is it possible to have char x[5] in one file a declaration extern char \* in other file ?**

**15. What is dangling pointer ?**

**16. Why does the below code cause segmentation fault ?**

```
int* z = NULL;
```

```
*z = 1;
```

**17. What are the two problems in the below code ?**

```
char *s1 = "Hello, ";
```

```
char *s2 = "world!";
```

```
char *s3 = strcat(s1, s2);
```

**18. What is the problem with the below code ?**

i) `char a[] = "Hello";`

`char *p = "Hello";`

My program crashes if I try to assign a new value to `p[i]`.

ii) `char *a = "abcdef";`

```
*a = 'X';
```

**19. Some compilers have a switch to control if string literals are writable or not – True/False?**

**20. Three attributes can be assign to a pointer: const / volatile / restrict – True/False**

**21. What are the problems in below code. How will you fix it ?**

```
char *check(int n)
```

```
{
char box[20];
sprintf(box, "%d", n);
return box;
}
```

**22. What is malloc's internal bookkeeping information and how does it play a role in mis-directing the location of the actual bug ?**

**23. Where would you use 'const int' instead of #define and where should you use #define instead of 'const int' ?**

**24. Keyword 'const' refers to read-only -> True/False?**

**25. What is the output of the below code**

```
#define MY_CALCULATOR 2+5+6
printf("%d \n" 3 * MY_CALCULATOR);
```

**26. How does declaring function parameters as 'const' help in better,safer code ?**

**27. Which of the following is correct . Why does point no 'i' gives output sometime & sometimes it does not give output ?**

- i. char \*a = malloc(strlen(str));  
strcpy(a, str);
- ii. char \*a = malloc(strlen(str) + 1);  
strcpy(a, str);

**28. How do the below get expanded ? Which one is the preferred method & Why ?**

```
#define mydef struct s *
typedef struct s * mytype;

mydef d1,d2;

mytype t1,t2;
```

**29. i. const values cannot be used in initializers – True/False?**

**ii. const values cannot be used for array dimensions – True/False**

**30. Is char x[3] = "wow"; legal ?**

Why does it work sometimes ?

**31. How will the below code corrupt stack ?**

```
void checkstackcorruption (char *var)
{
 char z[12];
 memcpy(z, var, strlen(var));
}

int main (int argc, char **argv)
{
 checkstackcorruption (argv[1]);
}
```

**32. Is the below method of usage of realloc() correct ?**

```
void *z = malloc(10);
free(z);
z = realloc(z, 20);
```

**33. What does the below mean ?**

```
int (*)[*];
```

**34. The rank of any unsigned integer type shall equal the rank of the corresponding**

**signed integer type, if any – True/False?**

**35. The rank of long long int shall be greater than the rank of long int which shall be greater than int – True/False?**

**36.No two signed integer types shall have the same rank, even if they have the same representation – True/False?**

**37. sprintf function is equivalent to fprintf, except that the output is written into an array – True/False?**

**38. Incase of both sprintf and fprintf , A null character is written at the end of the characters written and that is not counted as part of the returned value – True/False?**

**39. Arithmetic underflow on a negative number results in negative zero – True/False?**

**40.Negative zero and positive zero compare equal under default numerical comparison – True/False?**

**41. ‘type cast’ should not be used to override a const or volatile declaration – True/False?**

**42. ‘sizeof’ yields the size of an array in bytes – True / False?**

**43. How will you determine the number of elements in an array using sizeof operator ?**

**44. What is l-value & r-value ?**

**45. What is the output of the below code ?**

```
char (*x)[50];
printf("%u, %u,%d\n",x,x+1,sizeof(*x));
```

**46. How will you declare & initialize pointer to a register located at phys addrs 600000h ?**

**47. What is NPC ?**

**48. Why does \*ps.i do not work in the below code ?**

```
struct rec
{
int i;
float f;
char c;
};
int main()
{
struct rec *ps;
ps=(struct rec *) malloc (sizeof(struct rec));
*ps.i=10;
free ps;
return 0;
}
```

**49. The term ‘Pointer to pointer’ is different from the term ‘double pointer’ – True/False?**

**50. Will the Free API (after calling Malloc) return the Memory back to the OS or the Application ?**

**51. How to do array assignment ?**

**52. What could be the good way to check ‘Close enough’ equality using Floats ?**

**53. What is the Problem of Floating Points with ‘==’ operator in C . On what is that dependent upon ?**

**54. Tell about the size of ‘empty structs’ in C ?**

55. Why does C language permits an extra comma in initializer list
56. What does the below do ?  $z = x++ + (y += x++) ;$
57. What is nested union ? How is it useful ?
58. How will you pass unions to functions or pointers to unions ?
59. What is the Behaviour of realloc for NULL argument ?
60. Write a Macro to swap 2 bytes
61. Write a Macro to swap 2 bytes
62. Write small program to Set a Bit, Clear a Bit, Toggle a Bit, Test a Bit (or Demonstrate simple Bit Manipulation in C)
63. Use #define to Set Bit, Clr Bit, Toggle and Test Bit of a volatile status register (Port) – Bit Manipulations to set/clear a bit of a particular Port Register
64. Universal Gates;

Ans: <http://www.asic-world.com/digital/gates3.html>

### Memory Map In C

#### Memory Map in C

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" (Block Started by Symbol) segment, named after an ancient assembler operator that stood for "block started by symbol." Data in this segment is initialized by the kernel to arithmetic 0 before the program starts executing**

**uninitialized data starts at the end of the data segment and contains all global variables and static variables that are initialized to zero or do not have explicit initialization in source code.**

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

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

**4. Stack:**

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

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

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

**5. Heap:**

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

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

**Examples.**

The size(1) command reports the sizes (in bytes) of the text, data, and bss segments. ( for more details please refer man page of size(1) )

**1. Check the following simple C program**

```
#include <stdio.h>int main(void){return 0;}
```

```
[narendra@CentOS]$ gcc memory-layout.c -o memory-layout
```

```
[narendra@CentOS]$ size memory-layout
```

| text | data | bss | dec  | hex | filename      |
|------|------|-----|------|-----|---------------|
| 960  | 248  | 8   | 1216 | 4c0 | memory-layout |

**2. Let us add one global variable in program, now check the size of bss (highlighted in red color).**

```
#include <stdio.h>int global; /* Uninitialized variable stored in bss*/int main(void){return 0;}
```

[narendra@CentOS]\$ gcc memory-layout.c -o memory-layout

[narendra@CentOS]\$ size memory-layout

| text | data | bss | dec  | hex | filename      |
|------|------|-----|------|-----|---------------|
| 960  | 248  | 12  | 1220 | 4c4 | memory-layout |

**3. Let us add one static variable which is also stored in bss.**

```
#include <stdio.h>int global; /* Uninitialized variable stored in bss*/int main(void){static int i; /* Uninitialized static variable stored in bss */return 0;}
```

[narendra@CentOS]\$ gcc memory-layout.c -o memory-layout

[narendra@CentOS]\$ size memory-layout

| text | data | bss | dec  | hex | filename      |
|------|------|-----|------|-----|---------------|
| 960  | 248  | 16  | 1224 | 4c8 | memory-layout |

**4. Let us initialize the static variable which will then be stored in Data Segment (DS)**

```
#include <stdio.h>int global; /* Uninitialized variable stored in bss*/int main(void){static int i = 100; /* Initialized static variable stored in DS*/return 0;}
```

[narendra@CentOS]\$ gcc memory-layout.c -o memory-layout

[narendra@CentOS]\$ size memory-layout

| text | data | bss | dec  | hex | filename      |
|------|------|-----|------|-----|---------------|
| 960  | 252  | 12  | 1224 | 4c8 | memory-layout |

**5. Let us initialize the global variable which will then be stored in Data Segment (DS)**

```
#include <stdio.h>int global = 10; /* initialized global variable stored in DS*/int main(void){static int i = 100; /* Initialized static variable stored in DS*/return 0;}
```

[narendra@CentOS]\$ gcc memory-layout.c -o memory-layout

[narendra@CentOS]\$ size memory-layout

| text | data | bss | dec  | hex | filename      |
|------|------|-----|------|-----|---------------|
| 960  | 256  | 8   | 1224 | 4c8 | memory-layout |

**1. Where are global local static extern variables stored?**

- Local Variables are stored in Stack. Register variables are stored in Register. Global & static variables are stored in data segment (BSS). The memory created dynamically are stored in Heap and the C program instructions get stored in code segment and the extern variables also stored in data segment

**2. What does BSS Segment store?**

- BSS segment stores the uninitialized global and static variables and initializes them to zero. I read that BSS segment doesn't consume memory, then where those it store these variables? You probably read that the BSS segment doesn't consume space *in the executable file on disk*. When the executable loaded, the BSS segment certainly *does* consume space in memory. Space is allocated and initialized to zero by the OS loader

**3. Global variable and Local variable-**

- Global variables once declared they can be used anywhere in the program i.e. even in many functions. If possible u can use the global variables in the different user defined header files as like in packages in java. On the other hand global variables values can be changed programmatically local variables are local to a functional and can't be used beyond that function.

**4. Static variable and Global variable?**

- Static variables once declared they remain the same in the entire program and those values can't be changed programmatically.

global variables: check above description

**ASSEMBLER, LINKER AND LOADER:**

Normally the C's program building process involves four stages and utilizes different 'tools' such as a preprocessor, compiler, assembler, and linker.

- At the end there should be a single executable file. Below are the stages that happen in order regardless of the operating system/compiler and graphically illustrated in Figure w.1.

**1. Preprocessing is the first pass of any C compilation. It processes include-files, conditional compilation instructions and macros.**

**2. Compilation is the second pass. It takes the output of the preprocessor, and the source code, and generates assembler source code.**

**3. Assembly is the third stage of compilation. It takes the assembly source code and produces an assembly listing with offsets. The assembler output is stored in an object file.**

**4. Linking is the final stage of compilation. It takes one or more object files or libraries as input and combines them to produce a single (usually executable) file. In doing so, it resolves references to external symbols, assigns final addresses to procedures/functions and variables, and revises code and data to reflect new addresses (a process called relocation).**

- Bear in mind that if you use the IDE type compilers, these processes quite transparent.

- Now we are going to examine more details about the process that happen before and after the linking stage. For any given input file, the file name suffix (file extension)

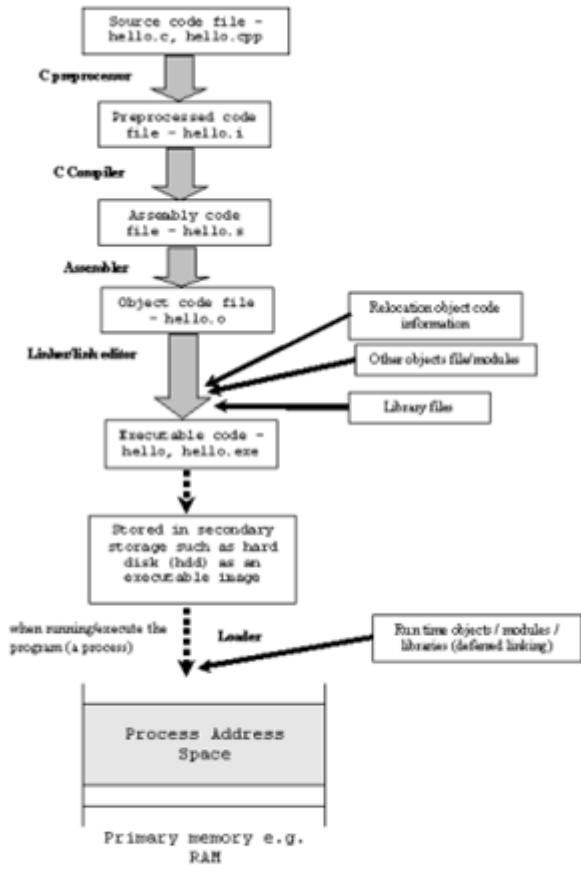
**determines what kind of compilation is done and the example for GCC is listed in Table w.1.**

- In UNIX/Linux, the executable or binary file doesn't have extension whereas in Windows the executables for example may have .exe, .com and .dll.**

| File extension                                | Description                                                                                                                             |
|-----------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------|
| file_name.c                                   | C source code which must be preprocessed.                                                                                               |
| file_name.i                                   | C source code which should not be preprocessed.                                                                                         |
| file_name.ii                                  | C++ source code which should not be preprocessed.                                                                                       |
| file_name.h                                   | C header file (not to be compiled or lin                                                                                                |
| file_name.ccx<br>file_name.cpp<br>file_name.C | C++ source code which must be preprocessed. For file_name.cxx, the xx must both be literally character x and file_name.C, is capital c. |
| file_name.s                                   | Assembler code.                                                                                                                         |
| file_name.S                                   | Assembler code which must be preprocessed.                                                                                              |
| file_name.o                                   | Object file by default, the object file name for a source file is made by replacing the extension .c, .i, .s etc with .o                |

Table w.1

**The following Figure shows the steps involved in the process of building the C program starting from the compilation until the loading of the executable image into the memory for program running.**



**Figure w.1: Compile, link & execute stages for running program**

## W.2 OBJECT FILES and EXECUTABLE

- After the source code has been assembled, it will produce an Object files (e.g. .o, .obj) and then linked, producing an executable files.
- An object and executable come in several formats such as ELF (Executable and Linking Format) and COFF(Common Object-File Format). For example, ELF is used on Linux systems, while COFF is used on Windows systems.
- Other object file formats are listed in the following Table.

| Object File Format | Description                                                                                                                                                                                                                                                                                                   |
|--------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| a.out              | The a.out format is the original file format for Unix. It consists of three sections: text, data, and bss, which are for program code, initialized data, and uninitialized data, respectively. This format is so simple that it doesn't have any reserved place for debugging information. The only debugging |

|          |                                                                                                                                                                                                                                                                                                                                                   |
|----------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|          | format for a.out is stabs, which is encoded as a set of normal symbols with distinctive attributes.                                                                                                                                                                                                                                               |
| COFF     | The COFF (Common Object File Format) format was introduced with System V Release 3 (SVR3) Unix. COFF files may have multiple sections, each prefixed by a header. The number of sections is limited. The COFF specification includes support for debugging but the debugging information was limited. There is no file extension for this format. |
| ECOFF    | A variant of COFF. ECOFF is an Extended COFF originally introduced for Mips and Alpha workstations.                                                                                                                                                                                                                                               |
| XCOFF    | The IBM RS/6000 running AIX uses an object file format called XCOFF (eXtended COFF). The COFF sections, symbols, and line numbers are used, but debugging symbols are dbx-style stabs whose strings are located in the .debug section (rather than the string table). The default name for an XCOFF executable file is a.out.                     |
| PE       | Windows 9x and NT use the PE ( <b>Portable Executable</b> ) format for their executables. PE is basically COFF with additional headers. The extension normally .exe.                                                                                                                                                                              |
| ELF      | The ELF (Executable and Linking Format) format came with System V Release 4 (SVR4) Unix. ELF is similar to COFF in being organized into a number of sections, but it removes many of COFF's limitations. ELF used on most modern Unix systems, including GNU/Linux, Solaris and Irix. Also used on many embedded systems.                         |
| SOM/ESOM | SOM (System Object Module) and ESOM (Extended SOM) is HP's object file and debug format (not to be confused with IBM's SOM, which is a cross-language Application Binary Interface – ABI).                                                                                                                                                        |

Table w.2

- When we examine the content of these object files there are areas called sections. Sections can hold executable code, data, dynamic linking information, debugging data, symbol tables, relocation information, comments, string tables, and notes.
- Some sections are loaded into the process image and some provide information needed in the building of a process image while still others are used only in linking object files.
- There are several sections that are common to all executable formats (may be named differently, depending on the compiler/linker) as listed below:

| Section | Description                                                                                                                                |
|---------|--------------------------------------------------------------------------------------------------------------------------------------------|
| .text   | This section contains the executable instruction codes and is shared among every process running the same binary. This section usually has |

|                                        |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |
|----------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|                                        | READ and EXECUTE permissions only. This section is the one most affected by optimization.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
| .bss                                   | BSS stands for 'Block Started by Symbol'. It holds un-initialized global and static variables. Since the BSS only holds variables that don't have any values yet, it doesn't actually need to store the image of these variables. The size that BSS will require at runtime is recorded in the object file, but the BSS (unlike the data section) doesn't take up any actual space in the object file.                                                                                                                                                                                                            |
| .data                                  | Contains the initialized global and static variables and their values. It is usually the largest part of the executable. It usually has READ/WRITE permissions.                                                                                                                                                                                                                                                                                                                                                                                                                                                   |
| .rdata                                 | Also known as .rodata (read-only data) section. This contains constants and string literals.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
| .reloc                                 | Stores the information required for relocating the image while loading.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| Symbol table                           | A symbol is basically a name and an address. Symbol table holds information needed to locate and relocate a program's symbolic definitions and references. A symbol table index is a subscript into this array. Index 0 both designates the first entry in the table and serves as the undefined symbol index. The symbol table contains an array of symbol entries.                                                                                                                                                                                                                                              |
| Relocation records                     | Relocation is the process of connecting symbolic references with symbolic definitions. For example, when a program calls a function, the associated call instruction must transfer control to the proper destination address at execution. Re-locatable files must have relocation entries' which are necessary because they contain information that describes how to modify their section contents, thus allowing executable and shared object files to hold the right information for a process's program image. Simply said relocation records are information used by the linker to adjust section contents. |
| Table w.3: Segments in executable file |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |

- The following is an example of the object file content dumping using readelf program. Other utility can be used is objdump.

```
/* testprog1.c */
#include <stdio.h>

static void display(int i, int *ptr);

int main(void)
{
 int x = 5;
```

```

int *xptr = &x;
printf("In main() program:\n");
printf("x value is %d and is stored at address %p.\n", x, &x);
 printf("xptr pointer points to address %p which holds a value of %d.\n", xptr,
*xptr);
display(x, xptr);
return 0;
}

void display(int y, int *yptr)
{
 char var[7] = "ABCDEF";
 printf("In display() function:\n");
 printf("y value is %d and is stored at address %p.\n", y, &y);
 printf("yptr pointer points to address %p which holds a value of %d.\n", yptr,
*yptr);
}

```

[bodo@bakawali test]\$ gcc -c testprog1.c  
[bodo@bakawali test]\$ readelf -a testprog1.o

**ELF Header:****Magic: 7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00****Class: ELF32****Data: 2's complement, little endian****Version: 1 (current)****OS/ABI: UNIX – System V****ABI Version: 0****Type: REL (Relocatable file)****Machine: Intel 80386****Version: 0x1****Entry point address: 0x0****Start of program headers: 0 (bytes into file)****Start of section headers: 672 (bytes into file)**

**Flags:** 0x0

**Size of this header:** 52 (bytes)

**Size of program headers:** 0 (bytes)

**Number of program headers:** 0

**Size of section headers:** 40 (bytes)

**Number of section headers:** 11

**Section header string table index:** 8

#### Section Headers:

| [Nr] | Name            | Type     | Addr     | Off      | Size     | ES | Flg | Lk | Inf | Al |
|------|-----------------|----------|----------|----------|----------|----|-----|----|-----|----|
| [ 0] | NULL            | PROGBITS | 00000000 | 00000000 | 00000000 | 00 | 0   | 0  | 0   | 0  |
| [ 1] | .text           | PROGBITS | 00000000 | 000034   | 0000de   | 00 | AX  | 0  | 0   | 4  |
| [ 2] | .rel.text       | REL      | 00000000 | 00052c   | 000068   | 08 | 9   | 1  | 4   |    |
| [ 3] | .data           | PROGBIT  | 00000000 | 000114   | 00000000 | 00 |     | WA | 0   | 0  |
| [ 4] | .bss            | NOBIT    | 00000000 | 000114   | 00000000 | 00 | WA  | 0  | 0   | 4  |
| [ 5] | .rodata         | PROGBITS | 00000000 | 000114   | 00010a   | 00 | A   | 0  | 0   | 4  |
| [ 6] | .note.GNU-stack | PROGBITS | 00000000 | 00021e   | 00000000 | 00 | 0   | 0  | 1   |    |
| [ 7] | .comment        | PROGBITS | 00000000 | 00021e   | 000031   | 00 | 0   | 0  | 1   |    |
| [ 8] | .shstrtab       | STRTAB   | 00000000 | 00024f   | 000051   | 00 | 0   | 0  | 1   |    |
| [ 9] | .symtab         | SYMTAB   | 00000000 | 000458   | 0000b0   | 10 | 10  | 9  | 4   |    |
| [10] | .strtab         | STRTAB   | 00000000 | 000508   | 000021   | 00 | 0   | 0  | 1   |    |

#### Key to Flags:

**W** (write), **A** (alloc), **X** (execute), **M** (merge), **S** (strings)

**I** (info), **L** (link order), **G** (group), **x** (unknown)

**O** (extra OS processing required) **o** (OS specific), **p** (processor specific)

There are no program headers in this file.

Relocation section '.rel.text' at offset 0x52c contains 13 entries:

| Offset   | Info     | Type     | Sym.     | Value | Sym. | Name   |
|----------|----------|----------|----------|-------|------|--------|
| 0000002d | 00000501 | R_386_32 | 00000000 |       | .    | rodata |

```

00000032 00000a02 R_386_PC32 00000000 printf
00000044 00000501 R_386_32 00000000 .rodata
00000049 00000a02 R_386_PC32 00000000 printf
0000005c 00000501 R_386_32 00000000 .rodata
00000061 00000a02 R_386_PC32 00000000 printf
0000008c 00000501 R_386_32 00000000 .rodata
0000009c 00000501 R_386_32 00000000 .rodata
000000a1 00000a02 R_386_PC32 00000000 printf
000000b3 00000501 R_386_32 00000000 .rodata
000000b8 00000a02 R_386_PC32 00000000 printf
000000cb 00000501 R_386_32 00000000 .rodata
000000d0 00000a02 R_386_PC32 00000000 printf

```

**There are no unwind sections in this file.**

**Symbol table '.syms' contains 11 entries:**

| Num:         | Value | Size    | Type   | Bind    | Vis | Ndx         | Name   |
|--------------|-------|---------|--------|---------|-----|-------------|--------|
| 0: 00000000  | 0     | NOTYPE  | LOCAL  | DEFAULT | UND |             |        |
| 1: 00000000  | 0     | FILE    | LOCAL  | DEFAULT | ABS | testprog1.c |        |
| 2: 00000000  | 0     | SECTION | LOCAL  | DEFAULT |     | 1           |        |
| 3: 00000000  | 0     | SECTION | LOCAL  | DEFAULT |     | 3           |        |
| 4: 00000000  | 0     | SECTION | LOCAL  | DEFAULT |     | 4           |        |
| 5: 00000000  | 0     | SECTION | LOCAL  | DEFAULT |     | 5           |        |
| 6: 00000080  | 94    | FUNC    | LOCAL  | DEFAULT | 1   | display     |        |
| 7: 00000000  | 0     | SECTION | LOCAL  | DEFAULT |     | 6           |        |
| 8: 00000000  | 0     | SECTION | LOCAL  | DEFAULT |     | 7           |        |
| 9: 00000000  | 128   | FUNC    | GLOBAL | DEFAULT | 1   | main        |        |
| 10: 00000000 | 0     | NOTYPE  | GLOBAL | DEFAULT | UND |             | printf |

[Memory Map In C\(2\)](#)

- When writing a program using the assembly language it should be compatible with the sections in the assembler directives (x86) and the partial list that is interested to us is listed below:

|   | <b>Section</b>                    | <b>Description</b>                                                 |
|---|-----------------------------------|--------------------------------------------------------------------|
| 1 | Text (.section .text)             | Contain code (instructions).Contain the <code>_start</code> label. |
| 2 | Read-Only Data (.section .rodata) | Contains pre-initialized constants.                                |
| 3 | Read-Write Data (.section .data)  | Contains pre-initialized variables.                                |
| 4 | BSS (.section .bss)               | Contains un-initialized data.                                      |

Table w.4

- The assembler directives in assembly programming can be used to identify code and data sections, allocate/initialize memory and making symbols externally visible or invisible.

- An example of the assembly code with some of the assembler directives (Intel) is shown below:

;initializing data

```
.section .data
x: .byte 128 ;one byte initialized to 128
y: .long 1,1000,10000 ;3 long words
```

;initializing ascii data

```
.ascii "hello" ;ascii without null character
asciz "hello" ;ascii with
```

;allocating memory in bss

```
.section .bss
.equ BUFSIZE 1024 ;define a constant
.comm z, 4, 4 ;allocate 4 bytes for x with;4-byte alignment
```

;making symbols externally visible

```
.section .data
```

```
.globl w ;declare externally visible;e.g: int w = 10
.text
.globl fool ;e.g: fool(void) {...}

fool:
...
leave
return
```

### W.3 RELOCATION RECORDS

- Because the various object files will include references to each others code and/or data, so various locations, these shall need to be combined during the link time.
- For example in Figure w.2, the object file that has `main()` includes calls to functions `funct()` and `printf()`.
- After linking all of the object files together, the linker uses the relocation records to find all of the addresses that need to be filled in.

### W.4 SYMBOL TABLE

- Since assembling to machine code removes all traces of labels from the code, the object file format has to keep these around in different places.
- It is accomplished by the symbol table that contains a list of names and their corresponding offsets in the text and data segments.
- A disassembler provides support for translating back from an object file or executable.

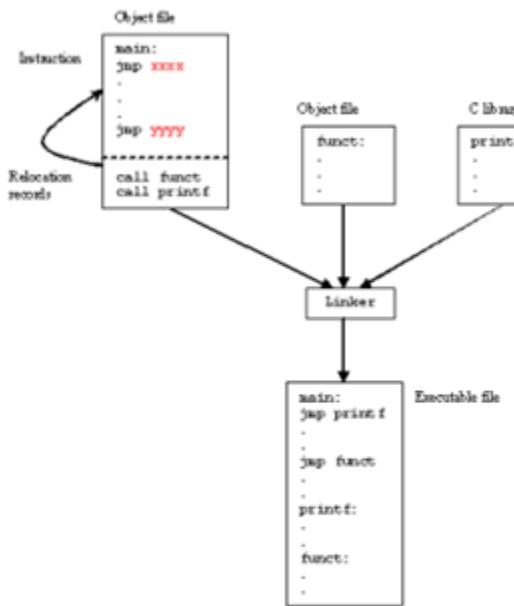


Figure w.2: The relocation record

## W.5 LINKING

The linker actually enables separate compilation. As shown in Figure w.3, an executable can be made up of a number of source files which can be compiled and assembled into their object files respectively, independently.

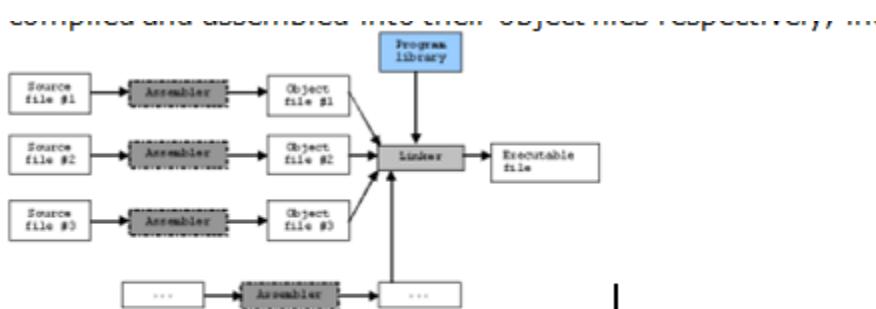


Figure w.3: The object files linking process

### W.5.1 SHARED OBJECTS

- In a typical system, a number of programs will be running. Each program relies on a number of functions, some of which will be standard C library functions, like `printf()`, `malloc()`, `strcpy()`, etc. and some are non-standard or user defined functions.
- If every program uses the standard C library, it means that each program would normally have a unique copy of this particular library present within it. Unfortunately, this result in wasted resources, degrade the efficiency and performance.
- Since the C library is common, it is better to have each program reference the common, one instance of that library, instead of having each program contain a copy of the library.
- This is implemented during the linking process where some of the objects are linked during the link time whereas some done during the run time (deferred/dynamic linking).

#### **W.5.2 STATICALLY LINKED**

- The term ‘statically linked’ means that the program and the particular library that it’s linked against are combined together by the linker at link time.
- This means that the binding between the program and the particular library is fixed and known at link time before the program run. It also means that we can’t change this binding, unless we re-link the program with a new version of the library.
- **Programs that are linked statically are linked against archives of objects (libraries) that typically have the extension of .a. An example of such a collection of objects is the standard C library, libc.a.**
- **You might consider linking a program statically for example, in cases where you weren’t sure whether the correct version of a library will be available at runtime, or if you were testing a new version of a library that you don’t yet want to install as shared.**
- **For gcc, the `-static` option can be used during the compilation/linking of the program.**

`gcc -static filename.c -o filename`

- The drawback of this technique is that the executable is quite big in size, all the needed information need to be brought together.

#### **W.5.3 DYNAMICALLY LINKED**

- The term ‘dynamically linked’ means that the program and the particular library it references are not combined together by the linker at link time.
- Instead, the linker places information into the executable that tells the loader which shared object module the code is in and which runtime linker should be used to find and bind the references.
- This means that the binding between the program and the shared object is done at runtime that is before the program starts, the appropriate shared objects are found and bound.
- This type of program is called a partially bound executable, because it isn’t fully resolved. The linker, at link time, didn’t cause all the referenced symbols in the program to be associated with specific code from the library.
- Instead, the linker simply said something like: “This program calls some functions within a particular shared object, so I’ll just make a note of which shared object these functions are in, and continue on”.

- Symbols for the shared objects are only verified for their validity to ensure that they do exist somewhere and are not yet combined into the program.
  - The linker stores in the executable program, the locations of the external libraries where it found the missing symbols. Effectively, this defers the binding until runtime.
  - Programs that are linked dynamically are linked against shared objects that have the extension .so. An example of such an object is the shared object version of the standard C library, libc.so.
  - The advantageous to defer some of the objects/modules during the static linking step until they are finally needed (during the run time) includes:
1. *Program files (on disk) become much smaller because they need not hold all necessary text and data segments information. It is very useful for portability.*
  2. *Standard libraries may be upgraded or patched without every one program need to be re-linked. This clearly requires some agreed module-naming convention that enables the dynamic linker to find the newest, installed module such as some version specification. Furthermore the distribution of the libraries is in binary form (no source), including dynamically linked libraries (DLLs) and when you change your program you only have to recompile the file that was changed.*
  3. *Software vendors need only provide the related libraries module required. Additional runtime linking functions allow such programs to programmatically-link the required modules only.*
  4. *In combination with virtual memory, dynamic linking permits two or more processes to share read-only executable modules such as standard C libraries. Using this technique, only one copy of a module needs be resident in memory at any time, and multiple processes, each can execute this shared code (read only). This results in a considerable memory saving, although demands an efficient swapping policy.*

## W.6 HOW SHARED OBJECTS ARE USED

- To understand how a program makes use of shared objects, let's first examine the format of an executable and the steps that occur when the program starts.

### W.6.1 SOME ELF FORMAT DETAILS

- Executable and Linking Format (ELF) is binary format, which is used in SVR4 Unix and Linux systems.
- It is a format for storing programs or fragments of programs on disk, created as a result of compiling and linking.
- ELF not only simplifies the task of making shared libraries, but also enhances dynamic loading of modules at runtime.

### W.6.2 ELF SECTIONS

- The Executable and Linking Format used by GNU/Linux and other operating systems, defines a number of 'sections' in an executable program.

- These sections are used to provide instruction to the binary file and allowing inspection. Important function sections include the Global Offset Table (GOT), which stores addresses of system functions, the Procedure Linking Table (PLT), which stores indirect links to the GOT, `.init/.fini`, for internal initialization and shutdown, `.ctors/.dtors`, for constructors and destructors.

- The data sections are `.rodata`, for read only data, `.data` for initialized data, and `.bss` for uninitialized data.

- Partial list of the ELF sections are organized as follows (from low to high):

1. `.init` – *Startup*
2. `.text` – *String*
3. `.fini` – *Shutdown*
4. `.rodata` – *Read Only*
5. `.data` – *Initialized Data*
6. `.tdata` – *Initialized Thread Data*
7. `.tbss` – *Uninitialized Thread Data*
8. `.ctors` – *Constructors*
9. `.dtors` – *Destructors*
10. `.got` – *Global Offset Table*
11. `.bss` – *Uninitialized Data*

- You can use the `readelf` or `objdump` program against the object or executable files in order to view the sections.

- In the following Figure, two views of an ELF file are shown: the linking view and the execution view.

**Figure w.4: Simplified object file format: linking view and execution view.**

- Keep in mind that the full format of the ELF contains many more items. As explained previously, the linking view, which is used when the program or library is linked, deals with sections within an object file.
- Sections contain the bulk of the object file information: data, instructions, relocation information, symbols, debugging information, etc.
- The execution view, which is used when the program runs, deals with segments. Segments are a way of grouping related sections.

- For example, the text segment groups executable code, the data segment groups the program data, and the dynamic segment groups information relevant to dynamic loading.
- Each segment consists of one or more sections. A process image is created by loading and interpreting segments.
- The operating system logically copies a file's segment to a virtual memory segment according to the information provided in the program header table. The OS can also use segments to create a shared memory resource.
- At link time, the program or library is built by merging together sections with similar attributes into segments.
- Typically, all the executable and read-only data sections are combined into a single text segment, while the data and BSS are combined into the data segment.
- These segments are normally called load segments, because they need to be loaded in memory at process creation. Other sections such as symbol information and debugging sections are merged into other, non-load segments.

## W.7 PROCESS LOADING

- In Linux processes loaded from a file system (using either the `execve()` or `spawn()` system calls) are in ELF format.
- If the file system is on a block-oriented device, the code and data are loaded into main memory.
- If the file system is memory mapped (e.g. ROM/Flash image), the code needn't be loaded into RAM, but may be executed in place.
- This approach makes all RAM available for data and stack, leaving the code in ROM or Flash. In all cases, if the same process is loaded more than once, its code will be shared.
- Before we can run an executable, firstly we have to load it into memory.
- This is done by the loader, which is generally part of the operating system. The loader does the following things (from other things):

1. **Memory and access validation** – Firstly, the OS system kernel reads in the program file's header information and does the validation for type, access permissions, memory requirement and its ability to run its instructions. It confirms that file is an executable image and calculates memory requirements.
2. **Process setup includes:**
  1. **Allocates primary memory for the program's execution.**
  2. **Copies address space from secondary to primary memory.**
  3. **Copies the .text and .data sections from the executable into primary memory.**
  4. **Copies program arguments (e.g., command line arguments) onto the stack.**
  5. **Initializes registers: sets the esp (stack pointer) to point to top of stack, clears the rest.**

**6. Jumps to start routine, which: copies `main()`'s arguments off of the stack, and jumps to `main()`.**

- Address space is memory space that contains program code, stack, and data segments or in other word, all data the program uses as it runs.
- The memory layout, consists of three segments (text, data, and stack), in simplified form is shown in Figure w.5.
- The dynamic data segment is also referred to as the heap, the place dynamically allocated memory (such as from `malloc()` and `new`) comes from. Dynamically allocated memory is memory allocated at run time instead of compile/link time.
- This organization enables any division of the dynamically allocated memory between the heap (explicitly) and the stack (implicitly). This explains why the stack grows downward and heap grows upward.

Memory Map In C (3)

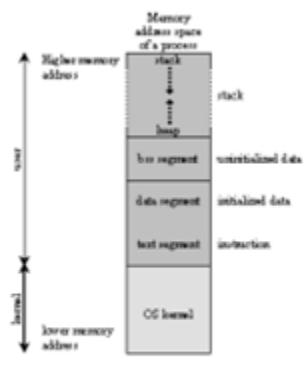


Figure w.4: Process memory layout

#### W.8 RUNTIME DATA STRUCTURE – From Sections to Segments

- A process is a running program. This means that the operating system has loaded the executable file for the program into memory, has arranged it to have access to its command-line arguments and environment variables, and has started it running.
- Typically a process has 5 different areas of memory allocated to it as listed in Table w.5 (refer to Figure w.4):

| Segment             | Description                                                                                                                                                                                                                                                                                                                         |
|---------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Code – text segment | Often referred to as the <b>text segment</b> , this is the area in which the executable instructions reside. For example, Linux/Unix arranges things so that multiple running instances of the same program share their code if possible. Only one copy of the instructions for the same program resides in memory at any time. The |

|                                        |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |
|----------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|                                        | portion of the executable file containing the text segment is the <b>text section</b> .                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
| Initialized data – <b>data</b> segment | Statically allocated and global data that are <b>initialized with nonzero values</b> live in the <b>data segment</b> . Each process running the same program has its own data segment. The portion of the executable file containing the data segment is the data section.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
| Uninitialized – <b>bss</b> segment     | <b>BSS</b> stands for ' <b>Block Started by Symbol</b> '. Global and statically allocated data that <b>initialized to zero by default</b> are kept in what is called the BSS area of the process. Each process running the same program has its own BSS area. When running, the BSS data are placed in the data segment. In the executable file, they are stored in the <b>BSS section</b> . For Linux/Unix the format of an executable, only variables that are initialized to a nonzero value occupy space in the executable's disk file.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |
| <b>Heap</b>                            | The heap is where dynamic memory (obtained by <code>malloc()</code> , <code>calloc()</code> , <code>realloc()</code> and <code>new</code> for C++) comes from. Everything on a heap is anonymous, thus you can only access parts of it through a pointer. As memory is allocated on the heap, the process's address space grows. Although it is possible to give memory back to the system and shrink a process's address space, this is almost never done because it will be allocated to other process again. Freed memory ( <code>free()</code> and <code>delete</code> ) goes back to the heap, creating what is called holes. It is typical for the heap to <b>grow upward</b> . This means that successive items that are added to the heap are added at addresses that are numerically greater than previous items. It is also typical for the heap to start immediately after the BSS area of the data segment. The end of the heap is marked by a pointer known as the <b>break</b> . You cannot reference past the break. You can, however, move the break pointer (via <code>brk()</code> and <code>sbrk()</code> system calls) to a new position to increase the amount of heap memory available. |
| <b>Stack</b>                           | The stack segment is where <b>local (automatic) variables are allocated</b> . In C program, local variables are all variables declared inside the opening left curly brace of a function body including the <code>main()</code> or other                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |

left curly brace that aren't defined as static. The data is popped up or pushed into the stack following the **Last In First Out**(LIFO) rule. The stack holds local variables, temporary information, function parameters, return address and the like. When a function is called, a **stack frame** (or a procedure activation record) is created and PUSHed onto the top of the stack. This stack frame contains information such as the address from which the function was called and where to jump back to when the function is finished (**return address**), parameters, local variables, and any other information needed by the invoked function. The order of the information may vary by system and compiler. When a function returns, the stack frame is POPped from the stack. Typically the **stack grows downward**, meaning that items deeper in the call chain are at numerically lower addresses and toward the heap.

Table w.5

- When a program is running, the initialized data, BSS and heap areas are usually placed into a single contiguous area called a data segment.
- The stack segment and code segment are separate from the data segment and from each other as illustrated in Figure w.4.
- Although it is theoretically possible for the stack and heap to grow into each other, the operating system prevents that event.
- The relationship among the different sections/segments is summarized in Table w.6, executable program segments and their locations.

| Executable file section<br>(disk file) | Address space segment | Program memory segment |
|----------------------------------------|-----------------------|------------------------|
| .text                                  | Text                  | Code                   |
| .data                                  | Data                  | Initialized data       |
| .bss                                   | Data                  | BSS                    |
| -                                      | Data                  | Heap                   |
| -                                      | Stack                 | Stack                  |

Table w.6

## W.9 THE PROCESS (IMAGE)

- The diagram below shows the memory layout of a typical C's process. The process load segments (corresponding to "text" and "data" in the diagram) at the process's base address.

- The main stack is located just below and grows downwards. Any additional threads or function calls that are created will have their own stacks, located below the main stack.
- Each of the stack frames is separated by a guard page to detect stack overflows among stacks frame. The heap is located above the process and grows upwards.
- In the middle of the process's address space, there is a region reserved for shared objects. When a new process is created, the process manager first maps the two segments from the executable into memory.
- It then decodes the program's ELF header. If the program header indicates that the executable was linked against a shared library, the process manager will extract the name of the dynamic interpreter from the program header.
- The dynamic interpreter points to a shared library that contains the runtime linker code. The process manager will load this shared library in memory and will then pass control to the runtime linker code in this library.

#### **W.10 RUNTIME LINKER AND SHARED LIBRARY LOADING**

- The runtime linker is invoked when a program that was linked against a shared object is started or when a program requests that a shared object be dynamically loaded.
- So the resolution of the symbols can be done at one of the following time:

1. *Load-time dynamic linking – the application program is read from the disk (disk file) into memory and unresolved references are located. The load time loader finds all necessary external symbols and alters all references to each symbol (all previously zeroed) to memory references relative to the beginning of the program.*
2. *Run-time dynamic linking – the application program is read from disk (disk file) into memory and unresolved references are left as invalid (typically zero). The first access of an invalid, unresolved, reference results in a software trap. The run-time dynamic linker determines why this trap occurred and seeks the necessary external symbol. Only this symbol is loaded into memory and linked into the calling program.*

- The runtime linker is contained within the C runtime library. The runtime linker performs several tasks when loading a shared library (.so file).
- The dynamic section provides information to the linker about other libraries that this library was linked against.
- It also gives information about the relocations that need to be applied and the external symbols that need to be resolved. The runtime linker will first load any other required shared libraries (which may themselves reference other shared libraries).
- It will then process the relocations for each library. Some of these relocations are local to the library, while others require the runtime linker to resolve a global symbol.
- In the latter case, the runtime linker will search through the list of libraries for this symbol. In ELF files, hash tables are used for the symbol lookup, so they're very fast.
- Once all relocations have been applied, any initialization functions that have been registered in the shared library's init section are called. This is used in some implementations of C++ to call global constructors.

#### **W.11 SYMBOL NAME RESOLUTION**

- When the runtime linker loads a shared library, the symbols within that library have to be resolved. Here, the order and the scope of the symbol resolution are important.

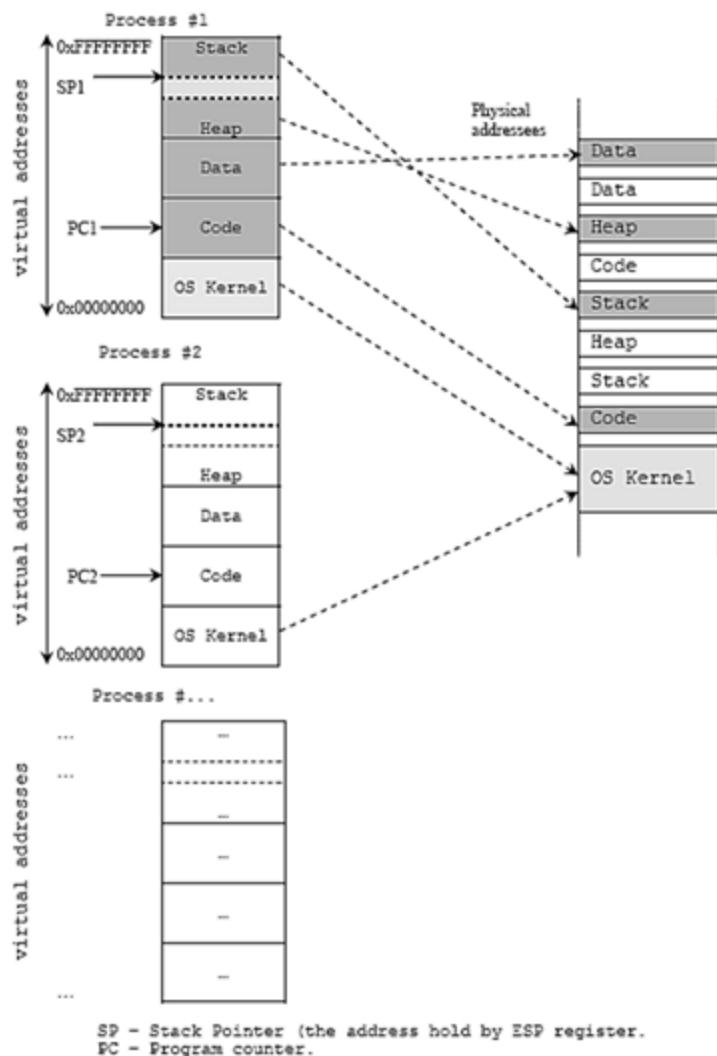
- If a shared library calls a function that happens to exist by the same name in several libraries that the program has loaded, the order in which these libraries are searched for this symbol is critical. This is why the OS defines several options that can be used when loading libraries.
- All the objects (executables and libraries) that have global scope are stored on an internal list (the global list).
- Any global-scope object, by default, makes available all of its symbols to any shared library that gets loaded.
- The global list initially contains the executable and any libraries that are loaded at the program's startup.

## W.12 DYNAMIC ADDRESS TRANSLATION

- In the view of the memory management, modern OS with multitasking, normally implement dynamic relocation instead of static.
- All the program layout in the address space is virtually same. This dynamic relocation (in processor term it is called dynamic address translation) provides the illusion that:

1. *Each process can use addresses starting at 0, even if other processes are running, or even if the same program is running more than one time.*
2. *Address spaces are protected.*
3. *Can fool process further into thinking it has memory that's much larger than available physical memory (virtual memory).*

- In dynamic relocation the address changed dynamically during every reference. Virtual address is generated by a process (also called logical address) and the physical address is the actual address in physical memory at the run-time.
- The address translation normally done by Memory Management Unit (MMU) that incorporated in the processor itself.
- Virtual addresses are relative to the process. Each process believes that its virtual addresses start from 0. The process does not even know where it is located in physical memory; the code executes entirely in terms of virtual addresses.
- MMU can refuse to translate virtual addresses that are outside the range of memory for the process for example by generating the segmentation faults. This provides the protection for each process.
- During translation, one can even move parts of the address space of a process between disk and memory as needed (normally called swapping or paging).
- This allows the virtual address space of the process to be much larger than the physical memory available to it.
- Graphically, this dynamic relocation for a process is shown in Figure w.6



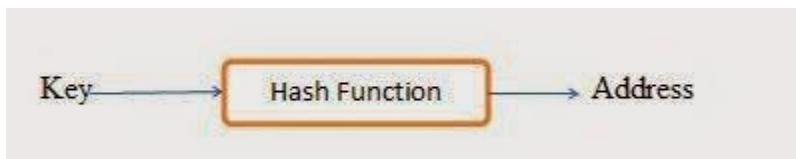
**Figure w.6: Physical and virtual address: Address translation**

### Data Structures Interview questions

What is hashing technique? Describe in brief.

**In general, in all searching techniques, search time is dependent on the number of items. Sequential search, binary search and all the search trees are totally dependent on number of items and many key comparisons are involved.**

**Hashing is a technique where search time is independent of the number of items or elements. In this technique a hash function is used to generate an address from a key. The hash function takes a key as input and returns the hash value of that key which is used as an address index in the array.**



We can write hash function as follows

$$h(k)=a;$$

Where **h** is hash function, **k** is the key, **a** is the hash value of the key.

While choosing a hash function we should consider some important points.

- It should be easy to compute
- It should generate address with minimum collision.

What are different techniques for making hash function? Explain with example.

**Techniques for making hash function.**

- Truncation Method
- Midsquare Method
- Folding Method
- Division Method

**Truncation Method**

This is the simplest method for computing address from a key. In this method we take only a part of the key as address.

**Example:**

Let us take some 8 digit keys and find addresses for them. Let the table size is 100 and we have to take 2 rightmost digits for getting the hash table address. Suppose the keys are. 62394572, 87135565, 93457271, 45393225.

So the address of above keys will be 72, 65, 71 and 25 respectively.

This method is easy to compute but chances of collision are more because last two digits can be same in more than one keys.

**Midsquare Method**

In this method the key is squared and some digits from the middle of this square are taken as address.

**Example:**

Suppose that table size is 1000 and keys are as follows

|               |         |         |         |         |
|---------------|---------|---------|---------|---------|
| Key           | 1123    | 2273    | 3139    | 3045    |
| Square of key | 1261129 | 5166529 | 9853321 | 9272025 |
| Address       | 612     | 665     | 533     | 720     |

### Folding Method

In this technique the key is divided into different part where the length of each part is same as that of the required address, except possibly the last part.

**Example:**

Let key is 123945234 and the table size is 1000 then we will broke this key as follows

123945234 ----> 123 945 234

Now we will add these broken parts.  $123+945+234=1302$ . The sum is 1302, we will ignore the final carry 1, so the address for the key 123945234 is 302.

### Division Method (Modulo-Division)

In Modulo-Division method the key is divided by the table size and the remainder is taken as the address of the hash table.

Let the table size is n then

$$H(k) = k \bmod n$$

**Example**

Let the keys are 123, 945, 234 and table size is 11 then the address of these keys will be.

$$123 \% 11 = 2$$

$$945 \% 11 = 10$$

$$235 \% 11 = 4$$

So the hash address of above keys will be 2,10,4.

**Note:** - Collisions can be minimized if the table size is taken to be a prime number.

What are different methods of collision resolution in hashing.

A collision occurs whenever a key is mapped to an address that is already occupied. Collision Resolution technique provides an alternate place in hash table where this key can be placed. Collision Resolution technique can be classified as:

**1) Open Addressing (Closed Hashing)**

**a) Linear Probing**

**b) Quadratic Probing**

**c) Double Hashing**

## 2) Separate Chaining (Open Hashing)

Describe Linear Probing with an example.

**In this method if address given by hash function is already occupied, then the key will be inserted in the next empty position in hash table. Let the table size is 7 and hash function returns address 4 for a key then we will search the empty location in this sequence.**

4, 5, 6, 7, 0, 1, 2, 3

**Example:**

Let the keys are 28, 47, 20, 36, 43, 23, 25, 54 and table size is 11 then

$$h(28)=28\%11=6$$

$$h(47)=47\%11=3$$

$$h(20)=20\%11=9$$

$$h(36)=36\%11=3$$

$$h(43)=43\%11=10$$

$$h(23)=23\%11=1$$

$$h(25)=25\%11=3$$

|    |    |
|----|----|
| 0  |    |
| 1  |    |
| 2  |    |
| 3  | 47 |
| 4  |    |
| 5  |    |
| 6  | 28 |
| 7  |    |
| 8  |    |
| 9  | 20 |
| 10 |    |

|    |    |
|----|----|
| 0  |    |
| 1  |    |
| 2  |    |
| 3  | 47 |
| 4  | 36 |
| 5  |    |
| 6  | 28 |
| 7  |    |
| 8  |    |
| 9  | 20 |
| 10 |    |

|    |    |
|----|----|
| 0  |    |
| 1  | 23 |
| 2  |    |
| 3  | 47 |
| 4  | 36 |
| 5  |    |
| 6  | 28 |
| 7  |    |
| 8  |    |
| 9  | 20 |
| 10 | 43 |

|    |    |
|----|----|
| 0  | 54 |
| 1  | 23 |
| 2  |    |
| 3  | 47 |
| 4  | 36 |
| 5  | 25 |
| 6  | 28 |
| 7  |    |
| 8  |    |
| 9  | 20 |
| 10 | 43 |

$$h(54)=54\%11=10 \text{ insert } 28, 47, 20$$

insert 36

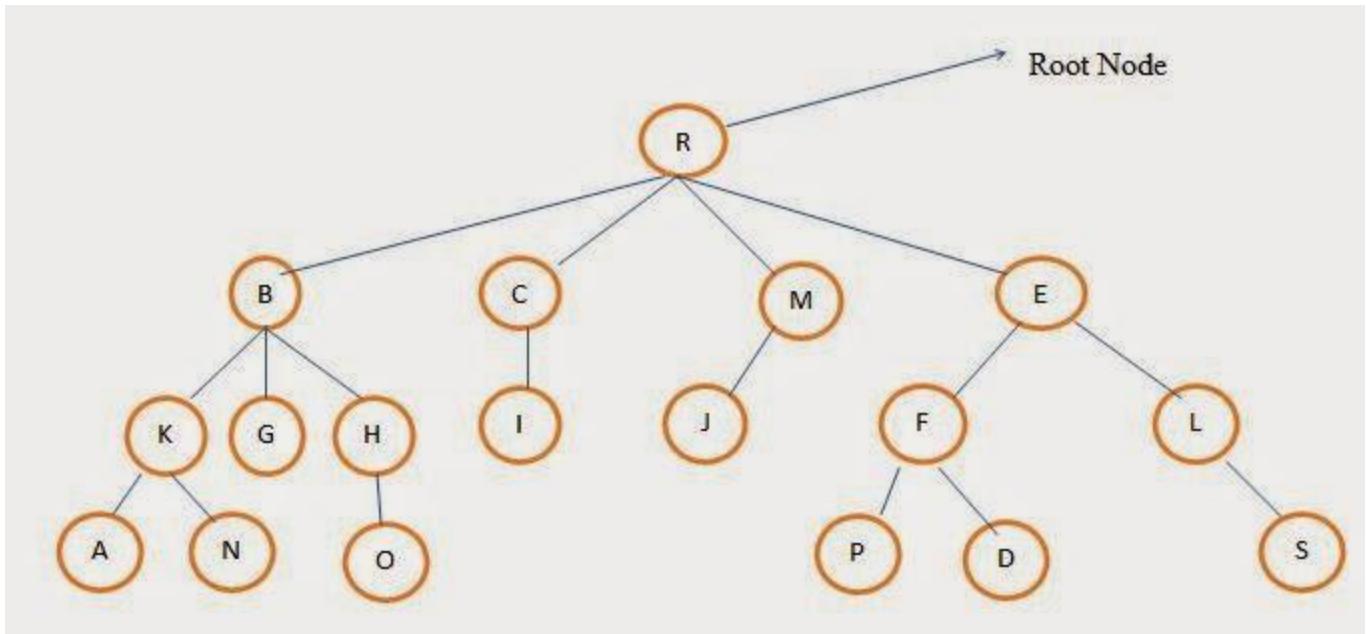
insert 43, 23

insert 25, 54

Describe the following term in a tree.

a) Level b) Height c) Degree.

Let's take an example to define the above term.

**Level:**

**Level of any node is defined as the distance of that node from the root. The level of root node is always zero. Node B, C, M, E are at level 1. Nodes K,G,H,I,J,F,L are at level 2. Nodes A,N,O,P,D,S are at level 3.**

**Height:**

**Height is also known as depth of the tree. The height of root node is one. Height of a tree is equal to one more than the largest level number of tree. The height of the above tree is 4.**

**Degree:**

**The number of children of a node is called its degree. The degree of node R is 4, the degree of node B is 3. The degree of node S is 0. The degree of a tree is the maximum degree of the node of the tree. The degree of the above given tree is 4.**

Describe binary tree and its property.

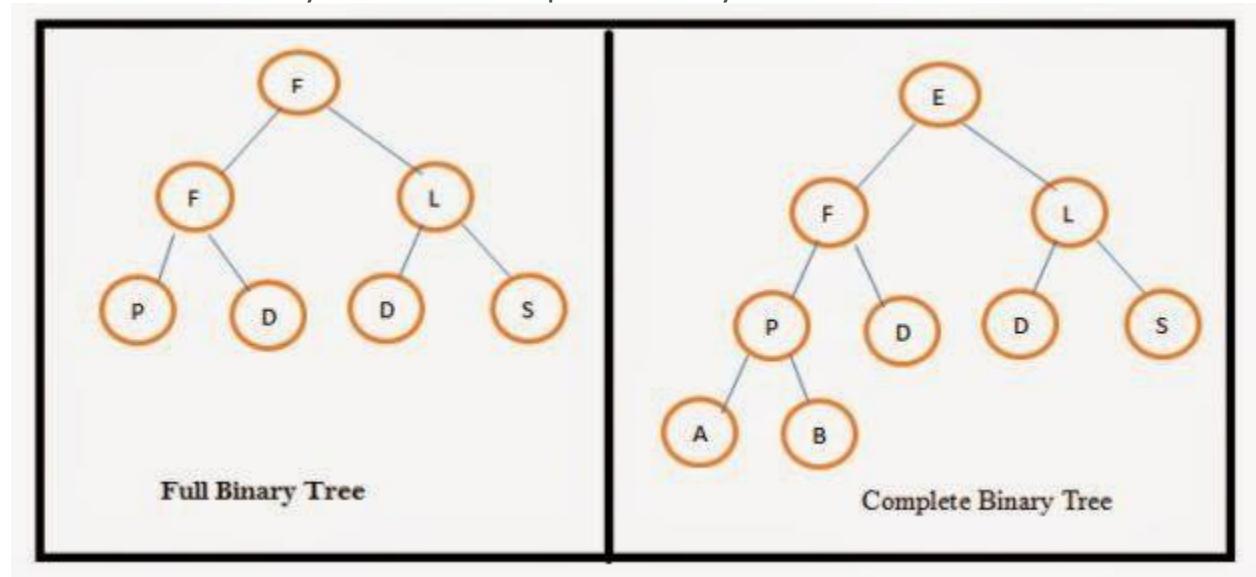
**In a binary tree a node can have maximum two children, or in other words we can say a node can have 0,1, or 2 children.**

**Properties of binary tree.**

- 1) **The maximum number of nodes on any level  $i$  is  $2^i$  where  $i \geq 0$ .**
- 2) **The maximum number of nodes possible in a binary tree of height  $h$  is  $2^h - 1$ .**
- 3) **The minimum number of nodes possible in a binary tree of height  $h$  is equal to  $h$ .**
- 4) **If a binary tree contains  $n$  nodes then its maximum possible height is  $n$  and minimum height possible is  $\log_2(n+1)$ .**
- 5) **If  $n$  is the total no of nodes and  $e$  is the total no of edges then  $e = n - 1$ . The tree must be non-empty binary tree.**

**6) If  $n_0$  is the number of nodes with no child and  $n_2$  is the number of nodes with 2 children, then  $n_0=n_2+1$ .**

Describe full binary tree and complete binary tree.



**Full binary tree:** A binary tree is full binary tree if all the levels have maximum number of nodes.

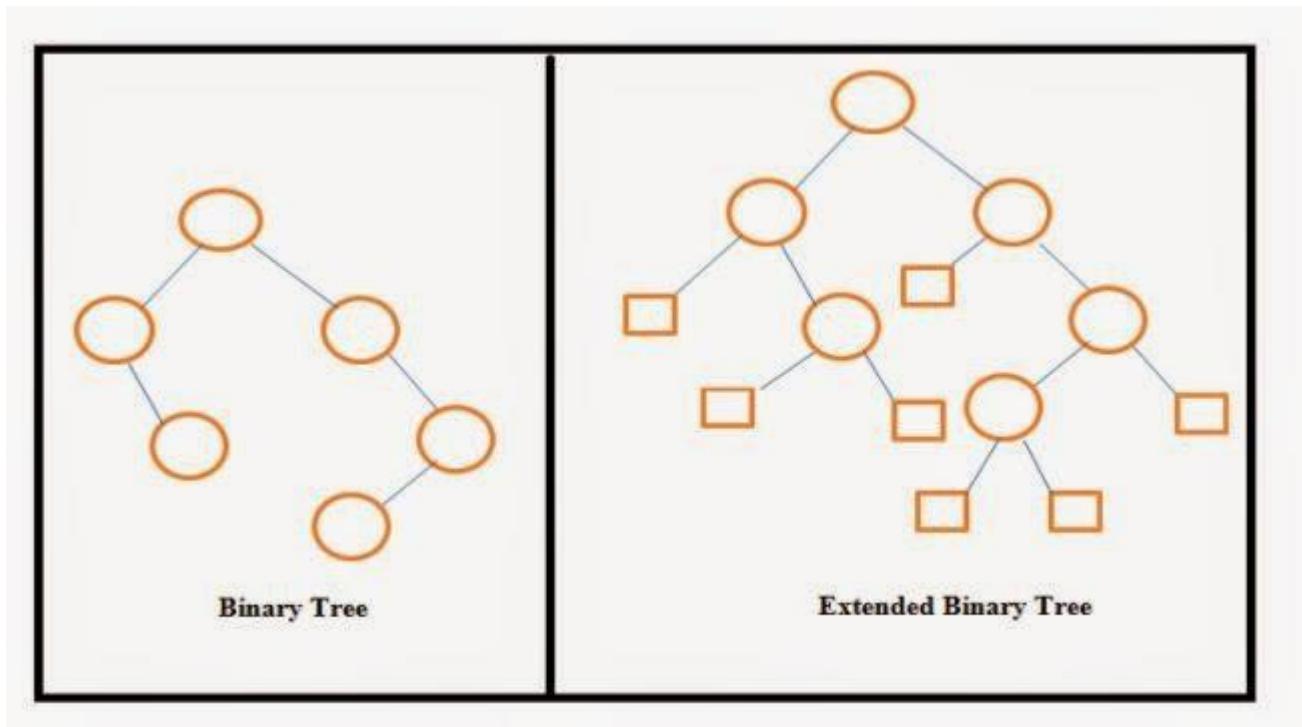
A full binary tree of height  $h$  has  $(2^h - 1)$  nodes.

**Complete binary tree:** In a complete binary tree all the levels have maximum number of nodes except possibly the last level.

The minimum no of nodes in a complete binary tree is  $2^{h-1}$  and the maximum number of nodes possible is  $(2^h - 1)$ . Where  $h$  is the height of the tree.

Explain Extended Binary tree.

A binarytree can be converted to an extended binary tree by adding special nodes to leaf nodes and nodes that have only one child. Extended binary tree is also called 2-tree.



In the above figure external nodes are shown by squares and internal nodes by circles. The extended binary tree is a strictly binary tree means each node has either 0 or 2 children. The path length of any node is the number of edges traversed from that node to the root node. Internal path length of a binary tree is the sum of path lengths of all internal nodes and external path length of a binary tree is the sum of path lengths of all external nodes.

What are different dynamic memory allocation technique in C .

The process of allocating memory at run time is called dynamic memory allocation. The allocation and release of this memory space can be done with the help of some predefined function. These functions allocate memory from a memory area called heap and free this memory whenever not required. The functions that are used to allocate memory at runtime are as follows:

- **malloc()**

- **calloc()**

- **realloc()**

### 1. **malloc()**

This function allocates memory dynamically. It is generally used as:

```
ptr= (datatype *) malloc(specified size);
```

Here ptr is a pointer of type datatype ( can be int, float, double....) and specified size is the size in bytes. The expression (datatype \*) is used to typecast the pointer returned by malloc().

**Example:**

```
int *ptr;
```

```
ptr=(int *)malloc(4*sizeof(int));
```

It allocates the memory space to hold four integer values and the address of first byte is stored in the pointer variable ptr. The allocated memory contains garbage value.

## 2. calloc()

The calloc() function is used to allocate multiple blocks of memory.

**Example:**

```
int *ptr;
```

```
ptr=(int *)calloc(4, sizeof(int));
```

It allocates 4 blocks of memory and each block contains 2 bytes.

## 3. realloc()

We can increase or decrease the memory allocated by malloc() or calloc() function. The realloc() function is used to change the size of the memory block. It changes the memory block without destroying the old data.

**Example:**

```
int *ptr;
```

```
ptr=(int *)malloc(4*sizeof(int));
```

```
ptr=(int *)realloc(ptr,newsize);
```

This function takes two argument, first is a pointer to the block of memory that was previously allocated by malloc() or calloc() and second argument is the new size of memory block.

```
ptr=(int *)realloc(ptr, 4*sizeof(int)); // newsize
```

Q.2 What are the difference between malloc() and calloc()?

**Following are the main difference between malloc() and calloc().**

- **calloc()** function takes two parameters but **malloc()** function takes only one parameter.

- **Memory allocated by calloc() is initialized to zero while memory allocated by malloc() contains garbage value.**

Q.3 How will you free the memory that is allocated at run time?

**Memory is one of the most important resources and it is limited. The dynamically allocated memory is not automatically released; it will exist till the end of program. So it is programmer's responsibility to free the memory after completion. The free() function is used to release the memory that is allocated at run time.**

**Example:**

```
free(ptr);
```

**Here ptr is a pointer variable that contains the base address of a memory block created by malloc() or calloc() function.**

Q.4 What are different application of stack.

**Some of the applications of stack are as follows:**

- **Function calls.**

- **Reversal of a string.**

- **Checking validity of an expression containing nested parenthesis.**

- **Conversion of infix expression to postfix.**

Q.5 How will you check the validity of an expression containing nested parentheses?

**One of the applications of stack is checking validity of an expression containing nested parenthesis. An expression will be valid if it satisfies the two conditions.**

- **The total number of left parenthesis should be equal to the total number of right parenthesis in the expression.**

- **For every right parenthesis there should be a left parenthesis of the same time.**

**The procedure for checking validity of an expression containing nested parenthesis:**

**1. First take an empty stack**

**2. Scan the symbols of expression from left to right.**

**3. If the symbol is a left parenthesis then push it on the stack.**

#### **4. If the symbol is right parenthesis then**

If the stack is empty then the expression is invalid because Right parentheses are more

**than left parenthesis.**

**else**

**Pop an element from stack.**

If popped parenthesis does not match the parenthesis being scanned then it is invalid

**because of mismatched parenthesis.**

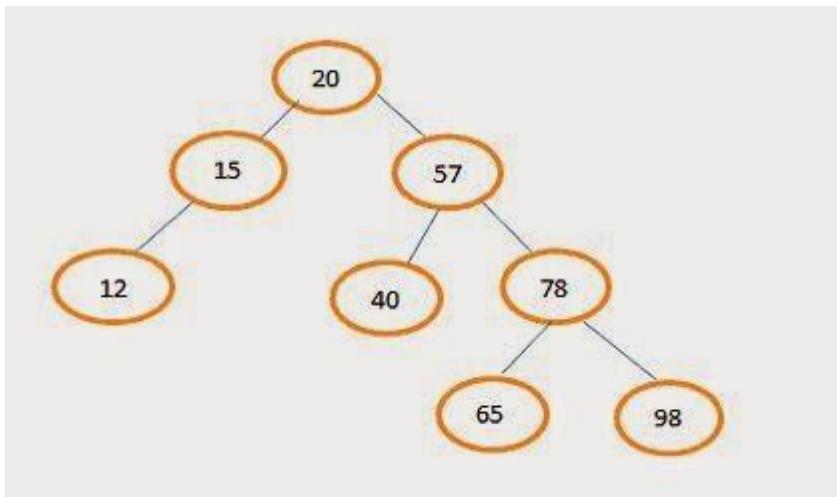
**5. After scanning all the symbols of expression, if stack is empty then expression is valid else it is invalid because left parenthesis is more than right parenthesis.**

Q.6 Give the example of validating the parenthesis of expression using stack.

Q.7 Describe AVL tree or height balanced binary search tree.

An AVL tree is binary search tree (BST) where the difference in the height of left and right subtrees of any node can be at most one. The technique for balancing the binary search tree was introduced by Russian Mathematicians G. M. Adelson and E. M. Landis in 1962. The height balanced binary search tree is called AVL tree in their honor.

## Example:



**For the leaf node 12, 40, 65 and 98 left and right subtrees are empty so difference of heights of their subtrees is zero.**

**For node 20 height of left subtree is 2 and height of right subtree is 3 so difference is 1.**

**For node 15 height of left subtree is 1 and height of right subtree is 0 so difference is 1.**

**For node 57 height of left subtree is 1 and height of right subtree is 2 so difference is 1.**

**For node 78 height of left subtree is 1 and height of right subtree is 1 so difference is 0.**

**Each node of an AVL tree has a balance factor, which is defined as the difference between the heights of left subtree and right subtree of a node.**

**Balance factor of a node=Height of its left subtree - Height of its right subtree.**

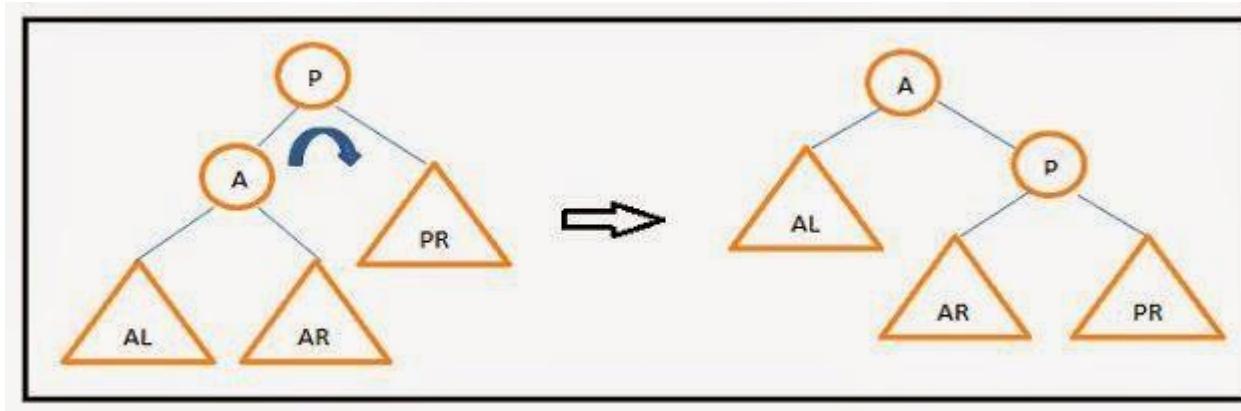
**In AVL tree possible values for the balance factor of any node are -1, 0, 1.**

**Q.8 Describe Tree Rotation in AVL tree.**

**After insertion or deletion operation the balance factor of the nodes in AVL tree can be changed and the tree may not be balanced. We can balance this tree by performing tree rotations. We know that AVL tree is binary search tree. Rotation of the tree should be in such a way that the new converted tree maintain the binary search tree property with inorder traversal same as that of the original tree. There are two types of tree rotation**

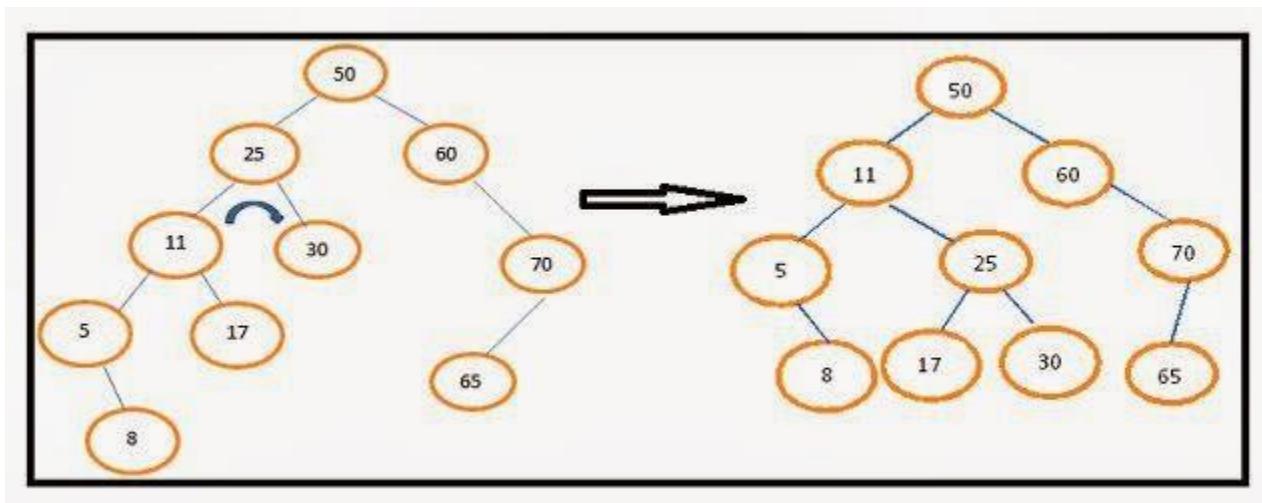
**- Right Rotation**

**- Left Rotation**

**Fig: Right Rotation**

Q.9 Give one example of Right Rotation.

**Right Rotation about node 25.**



### Data Structures Interview Questions

1. What is Data Structure?

**Data structure is a group of data elements grouped together under one name.**

**These data elements are called members. They can have different types and different lengths. Some of them store the data of same type while others store different types of data.**

2. Which data structure is used to perform recursion?

**The data structure used for recursion is Stack.**

**Its LIFO property helps it remembers its 'caller'. This helps it know the data which is to be returned when the function has to return.**

**System stack is used for storing the return addresses of the function calls.**

3. Does the Minimal Spanning tree of a graph give the shortest distance between any 2 specified nodes?

**No, it doesn't.**

**It assures that the total weight of the tree is kept to minimum.**

**It doesn't imply that the distance between any two nodes involved in the minimum-spanning tree is minimum.**

4. If you are using C language to implement the heterogeneous linked list, what pointer type will you use?

**A heterogeneous linked list contains different data types in its nodes. We can not use ordinary pointer to connect them.**

**The pointer that we use in such a case is void pointer as it is a generic pointer type and capable of storing pointer to any type.**

5. Differentiate between PUSH and POP?

**Pushing and popping refers to the way data is stored into and retrieved from a stack.**

**PUSH – Data being pushed/ added to the stack.**

**POP - Data being retrieved from the stack, particularly the topmost data.**

6. When is a binary search algorithm best applied?

**It is best applied to search a list when the elements are already in order or sorted.**

**The list here is searched starting in the middle. If that middle value is not the correct one, the lower or the upper half is searched in the similar way.**

7. How do you reference all the elements in a one-dimension array?  
**This is done using an indexed loop.**

**The counter runs from 0 to the array size minus one.**

**Using the loop counter as the array subscript helps in referencing all the elements in one-dimensional array.**

8. What is Huffman's algorithm?

**It is used in creating extended binary trees that have minimum weighted path lengths from the given weights.**

**It makes use of a table that contains frequency of occurrence for each data element.**

9. What is Fibonacci search?

**It is a search algorithm that applies to a sorted array.**

**It uses divide-and-conquer approach that reduces the time needed to reach the target element.**

10. Which data structure is applied when dealing with a recursive function?  
**A recursive function is a function that calls itself based on a terminating condition.**

**It uses stack.**

**Using LIFO, a call to a recursive function saves the return address. This tells the return address to the calling function after the call terminates.**

11. How does dynamic memory allocation help in managing data?

**Dynamic memory allocation helps to store simple structured data types.**

**It can combine separately allocated structured blocks to form composite structures that expand and contract as required.**

12. What is a bubble sort and how do you perform it?

**Bubble sort is a sorting technique which can be applied to data structures like arrays.**

**Here, the adjacent values are compared and their positions are exchanged if they are out of order.**

**The smaller value bubbles up to the top of the list, while the larger value sinks to the bottom.**

13. How does variable declaration affect memory allocation?

**The amount of memory to be allocated depends on the data type of the variable.**

**An integer type variable is needs 32 bits of memory storage to be reserved.**

14. You want to insert a new item in a binary search tree. How would you do it?

**Let us assume that the you want to insert is unique.**

**First of all, check if the tree is empty.**

**If it is empty, you can insert the new item in the root node.**

**If it is not empty, refer to the new item's key.**

**If the data to be entered is smaller than the root's key, insert it into the root's left subtree.**

**Otherwise, insert it into the root's right subtree.**

15. Why is the isEmpty() member method called?

**The isEmpty() member method is called during the dequeue process. It helps in ascertaining if there exists any item in the queue which needs to be removed.**

**This method is called by the dequeue() method before returning the front element.**

16. What is a queue ?

**A Queue refers to a sequential organization of data.**

**It is a FIFO type data structure in which an element is always inserted at the last position and any element is always removed from the first position.**

17. What is a deque?

**A deque is a double-ended queue.**

**The elements here can be inserted or removed from either end.**

18. What is a postfix expression?

**It is an expression in which each operator follows its operands.**

**Here, there is no need to group sub-expressions in parentheses or to consider operator precedence..**

### Data Structures Interview Questions

**What is a data structure? What are the types of data structures? Briefly explain them**

**The scheme of organizing related information is known as 'data structure'. The types of data structure are:**

**Lists: A group of similar items with connectivity to the previous or/and next data items.**

**Arrays: A set of homogeneous values**

**Records: A set of fields, where each field consists of data belongs to one data type.**

**Trees: A data structure where the data is organized in a hierarchical structure. This type of data structure follows the sorted order of insertion, deletion and modification of data items.**

**Tables: Data is persisted in the form of rows and columns. These are similar to records, where the result or manipulation of data is reflected for the whole table.**

**Define a linear and non linear data structure.**

**Linear data structure: A linear data structure traverses the data elements sequentially, in which only one data element can directly be reached. Ex: Arrays, Linked Lists**

**Non-Linear data structure: Every data item is attached to several other data items in a way that is specific for reflecting relationships. The data items are not arranged in a sequential structure. Ex: Trees, Graphs**

**Define in brief an array. What are the types of array operations?**

An array is a set of homogeneous elements. Every element is referred by an index.

Arrays are used for storing the data until the application expires in the main memory of the computer system. So that, the elements can be accessed at any time. The operations are:

- Adding elements
- Sorting elements
- Searching elements
- Re-arranging the elements
- Performing matrix operations
- Pre-fix and post-fix operations

**What is a matrix? Explain its uses with an example**

A matrix is a representation of certain rows and columns, to persist homogeneous data. It can also be called as double-dimensioned array.

Uses:

- To represent class hierarchy using Boolean square matrix
- For data encryption and decryption
- To represent traffic flow and plumbing in a network
- To implement graph theory of node representation

**Define an algorithm. What are the properties of an algorithm? What are the types of algorithms?**

Algorithm: A step by step process to get the solution for a well defined problem.

Properties of an algorithm:

- Should be written in simple English
- Should be unambiguous, precise and lucid
- Should provide the correct solutions
- Should have an end point
- The output statements should follow input, process instructions

- The initial statements should be of input statements
- Should have finite number of steps
- Every statement should be definitive

**Types of algorithms:**

- Simple recursive algorithms. Ex: Searching an element in a list
- Backtracking algorithms Ex: Depth-first recursive search in a tree
- Divide and conquer algorithms. Ex: Quick sort and merge sort
- Dynamic programming algorithms. Ex: Generation of Fibonacci series
- Greedy algorithms Ex: Counting currency
- Branch and bound algorithms. Ex: Travelling salesman (visiting each city once and minimize the total distance travelled)
- Brute force algorithms. Ex: Finding the best path for a travelling salesman
- Randomized algorithms. Ex. Using a random number to choose a pivot in quick sort).

### ***What is an iterative algorithm?***

The process of attempting for solving a problem which finds successive approximations for solution, starting from an initial guess. The result of repeated calculations is a sequence of approximate values for the quantities of interest.

### ***What is an recursive algorithm?***

Recursive algorithm is a method of simplification that divides the problem into sub-problems of the same nature. The result of one recursion is the input for the next recursion. The repetition is in the self-similar fashion. The algorithm calls itself with smaller input values and obtains the results by simply performing the operations on these smaller values. Generation of factorial, Fibonacci number series are the examples of recursive algorithms.

### ***Explain quick sort and merge sort algorithms.***

Quick sort employs the 'divide and conquer' concept by dividing the list of elements into two sub elements.

**The process is as follows:**

1. Select an element, pivot, from the list.
2. Rearrange the elements in the list, so that all elements those are less than the pivot are arranged before the pivot and all elements those are greater than the pivot are arranged after the pivot. Now the pivot is in its position.
3. Sort the both sub lists – sub list of the elements which are less than the pivot and the list of elements which are more than the pivot recursively.

**Merge Sort:** A comparison based sorting algorithm. The input order is preserved in the sorted output.

**Merge Sort algorithm is as follows:**

1. The length of the list is 0 or 1, and then it is considered as sorted.
2. Otherwise, divide the unsorted list into 2 lists each about half the size.
3. Sort each sub list recursively. Implement the step 2 until the two sub lists are sorted.
4. As a final step, combine (merge) both the lists back into one sorted list.

**What is Bubble Sort and Quick sort?**

**Bubble Sort:** The simplest sorting algorithm. It involves the sorting the list in a repetitive fashion. It compares two adjacent elements in the list, and swaps them if they are not in the designated order. It continues until there are no swaps needed. This is the signal for the list that is sorted. It is also called as comparison sort as it uses comparisons.

**Quick Sort:** The best sorting algorithm which implements the ‘divide and conquer’ concept. It first divides the list into two parts by picking an element a ‘pivot’. It then arranges the elements those are smaller than pivot into one sub list and the elements those are greater than pivot into one sub list by keeping the pivot in its original place.

**What are the difference between a stack and a Queue?**

**Stack – Represents the collection of elements in Last In First Out order.**

**Operations includes testing null stack, finding the top element in the stack, removal of top most element and adding elements on the top of the stack.**

**Queue - Represents the collection of elements in First In First Out order.**

**Operations include testing null queue, finding the next element, removal of elements and inserting the elements from the queue.**

**Insertion of elements is at the end of the queue**

**Deletion of elements is from the beginning of the queue.**

**Can a stack be described as a pointer? Explain.**

**A stack is represented as a pointer. The reason is that, it has a head pointer which points to the top of the stack. The stack operations are performed using the head pointer. Hence, the stack can be described as a pointer.**

***Explain the terms Base case, Recursive case, Binding Time, Run-Time Stack and Tail Recursion.***

**Base case:** A case in recursion, in which the answer is known when the termination for a recursive condition is to unwind back.

**Recursive Case:** A case which returns to the answer which is closer.

**Run-time Stack:** A run time stack used for saving the frame stack of a function when every recursion or every call occurs.

**Tail Recursion:** It is a situation where a single recursive call is consisted by a function, and it is the final statement to be executed. It can be replaced by iteration.

***Is it possible to insert different type of elements in a stack? How?***

Different elements can be inserted into a stack. This is possible by implementing union / structure data type. It is efficient to use union rather than structure, as only one item's memory is used at a time.

***Explain in brief a linked list.***

A linked list is a dynamic data structure. It consists of a sequence of data elements and a reference to the next record in the sequence. Stacks, queues, hash tables, linear equations, prefix and post fix operations. The order of linked items is different that of arrays. The insertion or deletion operations are constant in number.

***Explain the types of linked lists.***

The types of linked lists are:

**Singly linked list:** It has only head part and corresponding references to the next nodes.

**Doubly linked list:** A linked list which both head and tail parts, thus allowing the traversal in bi-directional fashion. Except the first node, the head node refers to the previous node.

**Circular linked list:** A linked list whose last node has reference to the first node.

***How would you sort a linked list?***

**Step 1:** Compare the current node in the unsorted list with every element in the rest of the list. If the current element is more than any other element go to step 2 otherwise go to step 3.

**Step 2:** Position the element with higher value after the position of the current element. Compare the next element. Go to step1 if an element exists, else stop the process.

**Step 3:** If the list is already in sorted order, insert the current node at the end of the list. Compare the next element, if any and go to step 1 or quit.

***What is sequential search? What is the average number of comparisons in a sequential search?***

**Sequential search:** Searching an element in an array, the search starts from the first element till the last element.

The average number of comparisons in a sequential search is  $(N+1)/2$  where N is the size of the array. If the element is in the 1st position, the number of comparisons will be 1 and if the element is in the last position, the number of comparisons will be N.

**What are binary search and Fibonacci search?**

**Binary Search:** Binary search is the process of locating an element in a sorted list. The search starts by dividing the list into two parts. The algorithm compares the median value. If the search element is less than the median value, the top list only will be searched, after finding the middle element of that list. The process continues until the element is found or the search in the top list is completed. The same process is continued for the bottom list, until the element is found or the search in the bottom list is completed. If an element is found that must be the median value.

**Fibonacci Search:** Fibonacci search is a process of searching a sorted array by utilizing divide and conquer algorithm. Fibonacci search examines locations whose addresses have lower dispersion. When the search element has non-uniform access memory storage, the Fibonacci search algorithm reduces the average time needed for accessing a storage location.

**What is the method to find the complexity of an algorithm?**

Complexity of an algorithm can be found out by analyzing the resources like memory, processor, etc. The computational time is also used to find the complexity of an algorithm. The running time through which the program is processed requires the function of the size of the input. The complexity is measured by number of steps that has to be executed for a particular input. The space complexity and the time complexity are the two main methods which allow the user to know the complexity of the algorithm and allow user to make it more optimized.

**What is the use of space complexity and time complexity?**

The space complexity defines the storage capacity for the input data. It defines the amount of memory that is required to run a program to completion. The complexity like this depends on the size of the input data and the function that is used for the input size 'n'.

The time complexity deals with the amount of time required by a program to complete the whole process of execution. The time complexity allows creating optimized code and allowing user to calculate it before writing their own functions. The time complexity can be made such that a program can be optimized on the basis of the chosen method.

**Write an algorithm to show the postfix expression with the input given as : a b + c d + \* f ? .**

**The postfix expression deals with the expression where the operators are being written after their operands. The evaluation of these operators is being done from left-to-right. The algorithm that is used to show the input of a b + c d + \* f ? is as follows:**

```
Stack is cleared for the use,
symbol = insert input character
while(not end of input)
{
if (symbol is an operand)
push in stack;
else
{
pop two operands from the stack;
result=op1 symbol op2;
push result in stack;
}
symbol = next input character;
}
return (pop (stack));
```

**Write an algorithm through which the inserting and deleting of elements can take place in circular queue?**

**Circular queues are very important as it allows the element to circulate. In this queue rear end reaches the end and the first element in the list will become the rear of that queue. In this queue the front element can only become the rear element if and only if front has moved forward in the array. The algorithm that is used to represent it is as follows:**

**insert(queue,n,front,rear,item)**

**This procedure inserts an element item into a queue.**

**1. If front = 1 and rear = n, or if front = rear**

**+ 1, then:**

**Write: overflow, and return**

**2. [find new value of rear]**

**If front = NULL , then : [queue initially empty.]**

**Set front = 1 and rear=1.**

**Else if rear = n, then:**

**Set rear=1.**

**Else:**

**Set rear = rear + 1.**

**[end of structure.]**

**3. Set queue[rear] = item. [this inserts new element.]**

**4.Return to the beginning.**

#### **41. How expression trees are gets represented in data structure?**

**Expression trees are made up of operands like constants and variable names. The nodes also contain the operators. The tree is a binary tree on which all the binary operations are being performed. It consists of all the nodes consisting of two children. It is possible for a node to consist of only one child and other operators that can be used to evaluate the expression tree. The operator looks at the root value that is obtained by recursive evaluation of the subtrees. The expression tree is being handled to show the relationship between the operator and the operand.**

#### **How can a binary tree be represented using the rotation?**

**Binary tree can have rotations and it can be done by inserting a node in the binary search tree. There is a balancing factor that is required and it will be calculated like height of left subtree-height of right subtree. Each node in this case has 0,1,-1 factor value and beyond that there will be no modification possible to be done in the tree. If the balancing factor comes out to be either +2 or -2 then the tree becomes unbalanced. The rotation allows the tree to be balanced and it can be done with the left rotation and right rotation. The rotation also helps the searching to be quite faster than using any other tree. Using the balance factor a tree can be managed and rotation can be applied to the whole system to adjust and balance the tree.**

### **What is the difference between B tree and Binary search tree?**

Binary tree consists of only fixed number of keys and children, whereas B tree consists of variable number of keys and children. Binary tree keys are stored in decreasing order, whereas B tree consists of the keys and children in non-decreasing order.

Binary tree doesn't consist of associated child properties whereas B tree consists of keys has an association with all the nodes with the keys that are less than or equal to the preceding key. Binary tree doesn't have minimization factor concept, whereas B tree has the concept of minimization factor where each node has minimum number of allowable children.

### **What is the minimization factor and time complexity of B-tree?**

The minimization factor of every node consists of minimum number of children and they should have at least  $n-1$  keys. There are possibilities that the root node might violate certain properties that are having fewer than  $n-1$  keys. By seeing this every node can have at most  $2n-1$  keys or  $2n$  children. The height of the  $n$  key B-tree can be found out by keeping the minimum degree  $n \leq 2$ , and height as  $h = \log_2 (n+1/2)$ . The time complexity of this will be given in Big O notation that is the amount of time required to run the code to the completion. The upper bound of the B-tree is provided by this and the lower bound can also be the same. So, mathematically

$O(g(n)) = \{f(n) : \text{there exists positive constants such that } 0=f$

$f(n)=cg(n) \text{ for all } n, n=n_0\}$ , we say "f is oh of g"

### **How does Threaded binary tree represented in Data Structure?**

Threaded binary tree consists of a node that is in the binary tree and having no left or right child. The child that comes in the end is also called as leaf node then if there is no child present, it will be represented by the null pointers. The space that is occupied by null entries can be used to store valuable information about the node that is consisting of the child and the root node information. There is a special pointer that is used to point to the higher node in the tree also called as ancestor. These pointers are termed as threads and the whole representation is called as threaded binary tree. The binary tree can be represented in in-order, pre-order and post-order form. Every node in this type of tree doesn't have a right child. This doesn't allow the recursion of the tree. The code that is used to show the implementation is:

```
struct NODE
{
 struct NODE *leftchild;
 int node_value;
 struct NODE *rightchild;
 struct NODE *thread;
}
```

**Explain the sorting algorithm that is most suitable to be used with single linked list?**

The sorting algorithm that is most suitable with the single link list is the simple insertion sort. This consists of an array and link of pointers, where the pointers are pointing to each of the element in the array. For example:  $I[i] = i + 1$  for  $0 \leq i < n-1$  and  $I[n-1] = -1$ .

The linear link list can be pointed by the external pointer which initialized it to 0. To insert the nth element in the list the traversing time gets reduced and until the list is being sorted out completely the process doesn't end. The array that has to be traversed  $x[k]$  to sort the element and put them in the list. If the sorting is done then it reduces the time of the insertion of an element and time for searching for a particular element at proper position.

**What are the standard ways in which a graph can be traversed?**

There are two standard ways through which a graph can be traversed and these are:

i. The depth-first traversal: allow the graph to be traversed from a given point and then goes to the other points. The starting position is being defined as it doesn't consist of any root so, there is a specific point that is chosen to begin the traversing. In this the visits takes place at each vertex and then recursive action is taken to visit all the vertices adjacent to the node that is being visited. The graph can consists of cycles, but there is always a condition that the vertex has to be visited only once.

ii. The breadth-first traversal: allow the graph to be traverse one level by another level. Breadth-first visits all the nodes from the depth 0 and it consists of a root. The vertex that has to be visited has to be specified that will be traversed. The length of the vertex has to be defined to find the shortest path to the given vertex. Breadth-first traverse the starting vertex and then all the vertices that is been adjacent to it.

**How helpful is abstract data type of data structures?**

Abstract data type allows the user to write the code without even worrying about the type of data being used. It is a tool that specifies the logical properties of the data type. ADT is a type consisting set of operations that are called as interface. This interface is the only mechanism through which the data type can be accessed. It defines the new type of instance that is been created by operating on different data types. There is always some additional information on which ADT acts upon. It specifies the instance of the creation time. The abstract data type can be declared as:

`LIST<data type> variable name;`

**How to sequentially represent max-heap?**

Max heap is also known as descending heap consisting of the objects in a heap list with some keys. It is of size n and will be of the form of complete binary tree that is also of

**nodes n. In this max-heap each node is less than or equal to the content of its parent. It represents the sequential complete binary tree with the formula to calculate as:**

**max[j] <= max[(j-1)/2] for 0 <= ((j-1)/2) < j <= n-1**

**Max-heap contains the root element as the highest element in the heap and from there the descending elements will be shown on the children node. It will also be traversed in an orderly manner and will be accessed by accessing the root first then their children nodes.**

### **Write an algorithm to show the reverse of link list?**

**Link list is a data structure that is commonly used in programming. It is the simplest form. In this each node consists of a child node and a reference to the next node i.e. a link to another node. In this the group of nodes represents a sequence that helps in traversing of the node in an easy manner. The program is given to show the reversing of the link list:**

```
reverse(struct node **st)
{
 struct node *p, *q, *r;
 p = *st;
 q = NULL;
 while(p != NULL)
 {
 r = q;
 q = p;
 p = p
 link;
 q
 link = r;
 }
 *st = q;
}
```

### **Write an algorithm to show various operations on ordered list and arrays**

**Ordered list is a container holding the sequence of objects. In this each object is having a unique position in a particular sequence. The operations that are being provided and performed on the ordered list include:**

**FindPosition:** is used to find the position of the object in an ordered list.

**Operator[]:** is used to access the position of the object in the ordered list.

**withdraw(Position&):** is used to remove the object from a given position in an ordered list.

**InsertAfter:** is used to insert an object after some other object or on some position in the ordered list.

**InsertBefore:** is used to insert the object in an ordered list at a defined position in an array of the ordered list.

### **Write a program to show the insertion and deletion of an element in an array using the position**

To insert the element or delete the element, there is a requirement to find out the exact location as array is a group of linear characters, so it is very important to find the position of the element in an array before performing the actions on them. The program explains the insertion and deletion operations performed on an array of elements:

```
void insert (int *arr, int pos, int num)
/* This inserts an element at a given position*/
{
int i ;
for (i = MAX - 1 ; i >= pos ; i--)
arr[i] = arr[i - 1];
arr[i] = num ;
// This tells about the shifting of an element to the right
}// function of the insertion ends here

void del (int *arr, int pos)
/* This function is used to delete the element at a given position*/
{
int i ;
for (i = pos ; i < MAX ; i++)
arr[i - 1] = arr[i];
arr[i - 1] = 0 ;
}
```

**Write an algorithm to show the procedure of insertion into a B-tree?**

The procedure or the algorithm to insert an element into a B-tree is as follows:

1. There is a search function that is applied to find the place where there is any new record to be present. When there is any key inserted then there is a sorting function that works to put the elements in the sorted order.
2. Then a particular node is selected and checked that is there any way to insert the new record. If there is any way to insert the new record then an appropriate pointer being given that remains one more than number of records.
3. If the node overflows due to the increase in the size of the node and upper bound, then there is a splitting being done to decrease the size of that particular node.
4. The split will occur till all the records are not accommodated properly at their place. There can be a way to split the root as well to create the provision for the new record data.

**Write an algorithm that counts number of nodes in the circular linked list**

Circular linked list is a list in which the insertion and deletion can be done in two ways. There is a provision to count the number of nodes just by keeping a count variable to count the data that is being inserted in the circular list. The algorithm is given to show the count of the nodes in the circular linked list.

Keep a circular header list in memory.

Keep a count to traverse the whole list and to keep a count of the data element

set COUNT: = 0

1. Set PTR: = LINK [START]. {Initializes the pointer PTR}
  2. Repeat steps 3, 4, 5 while PTR = START;
  3. COUNT = COUNT + 1
  4. Set PTR = LINK [PTR]. [PTR now points to next node]
- [End of step 2 loop]
5. Exit

**Why Boundary Tag Representation is used?**

Boundary tag representation is used to show the memory management using the data structures. The memory is allocated from a large area where there is free memory available. Memory management allows the use of segment and process of allocation of resources. This allows the reservation of block of 'n' bytes of memory and a space that is larger than the space of the system. Some tasks are performed like identifying and merging of free segments that has to be released. The process of identification is

simpler as it only allows the location of the preceding segment to be located and used. The identification is done to find out the neighbors using the “used/free” flags that are kept with the location. There has to be kept some information regarding the processes and the resources that has been allocated to the data element. It allows the finding of the segment that is to find both the segments (start and end) to be used as a part of boundary tag.

### What does Simulation of queues mean?

Simulation is a process of forming and abstract model of the real world situation. It allows understanding the effect of modification and all the situations that are related to the queues like its properties. It allows the arrangement of the simulation program in a systematic way so that events can occur on regular basis. This allows the transaction to be done on time, for example suppose there is a person p1 submitting a bill and there has to be a period of time that will be taken to process his request. If there is any free window then the person can submit the bill and spend less time in the queue. If there is no free availability then there will be a queue. The person has the option to choose the shortest queue to get his work done quickly, but he has to wait until all the previous transactions are processed. This is the way simulation of the queue works.

### C Unknown Facts

1. [How was C made?](#)
2. [Why are all OS built by C, C++, etc.? Why not Java?](#)
3. [Which programming languages are mostly used for robotics?](#)
4. [If C and C++ give the best performance, why do we still code in other languages?](#)
5. [Can I write computer programs in Sanskrit?](#)
6. [What does this mean in C?](#)
7. [Which language is best C, C++, Python or Java?](#)
8. [Why was the computer language following C called C++ and not C+?](#)
9. [Why do many C functions have an extra F?](#)
10. [What are the most amazing C programs?](#)
11. [Why is the C language so hard to write?](#)
12. [Why is malloc\(\) harmful in embedded systems](#)
13. [Is memcpy more efficient than copying element by element in for loop iteratively?](#)
14. [What exactly does typedef do in C?](#)
15. [What is the difference between int\\* p and int \\*p?](#)
16. [Why multiplying pointers is not valid in C?](#)
17. [How can I set variable value as infinity in C?](#)
18. [When should I give parameters to main\(\) in C](#)

---

## Embedded systems interview questions - Embedded systems FAQ

- [Other Programming Languages](#) >> Embedded systems interview questions - Embedded systems FAQ

- [Next Page »](#)

## What is the need for an infinite loop in Embedded systems?

- Infinite Loops are those program constructs where in there is no break statement so as to get out of the loop, it just keeps looping over the statements within the block defined.

### Example:

```
While(Boolean True) OR for(;;);
{
//Code
}
```

- Embedded systems need infinite loops for repeatedly processing/monitoring the state of the program. One example could be the case of a program state continuously being checked for any exceptional errors that might just occur during run time such as memory outage or divide by zero etc.,

- For e.g. Customer care Telephone systems where in a pre-recorded audio file is played in case the dialer is put on hold..

- Also circuits being responsible for indicating that a particular component is active/alive during its operation by means of LED's.

## How does combination of functions reduce memory requirements in embedded systems?

- The amount of code that has to be dealt with is reduced thus easing the overhead and redundancy is eliminated in case if there is anything common among the functions.

- Memory allocation is another aspect that is optimized and it also makes sense to group a set of functions related in some way as one single unit rather than having them to be dispersed in the whole program.

- In case of interactive systems display of menu list and reading in the choices of user's could be encapsulated as a single unit.

## A vast majority of High Performance Embedded systems today use RISC architecture why?

- According to the instruction sets used, computers are normally classified into RISC and CISC. RISC stands for 'Reduced Instruction Set Computing' . The design philosophy of RISC architecture is such that only one instruction is performed on each machine cycle thus taking very less time and speeding up when compared to their CISC counterparts.
- Here the use of registers is optimised as most of the memory access operations are limited to store and load operations.
- Fewer and simple addressing modes, and simple instruction formats leads to greater efficiency, optimisation of compilers, re-organisation of code for better throughput in terms of space and time complexities. All these features make it the choice of architecture in majority of the Embedded systems.
- CISC again have their own advantages and they are preferred whenever the performance and compiler simplification are the issues to be taken care of.

## Why do we need virtual device drivers when we have physical device drivers?

Device drivers are basically a set of modules/routines so as to handle a device for which a direct way of communication is not possible through the user's application program and these can be thought of as an interface thus keeping the system small providing for minimalistic of additions of code, if any.

Physical device drivers can't perform all the logical operations needed in a system in cases like IPC, Signals and so on...

The main reason for having virtual device drivers is to mimic the behaviour of certain hardware devices without it actually being present and these could be attributed to the high cost of the devices or the unavailability of such devices.

These basically create an illusion for the users as if they are using the actual hardware and enable them to carryout their simulation results.

Examples could be the use of virtual drivers in case of Network simulators,also the support of virtual device drivers in case a user runs an additional OS in a virtual box kind of a software.

## What is the need for DMA in ES?

- Direct memory access is mainly used to overcome the disadvantages of interrupt and program controlled I/O.
- DMA modules usually take the control over from the processor and perform the memory operations and this is mainly because to counteract the mismatch in the processing speeds of I/O units and the processor. This is comparatively faster.
- It is an important part of any embedded systems, and the reason for their use is that they can be used for bursty data transfers instead of single byte approaches.
- It has to wait for the system's resources such as the system bus in case it is already in control of it.

### What is Endianness of a system and how do different systems communicate with each other?

- Endianness basically refers to the ordering of the bytes within words or larger bytes of data treated as a single entity.
- When we consider several bytes of data say for instance 4 bytes of data, XYZQ the lower byte if stored in a Higher address and others in successively decreasing addresses, then it refers to the Big Endian and the vice versa of this refers to Little Endian architecture.
- Intel 80x86 usually follows Little Endian and others like IBM systems follow Big Endian formats.
- If the data is being transmitted care has to be taken so as to know as to which byte, whether the higher or the lower byte is being transmitted.
- Hence a common format prior to communication has to be agreed upon to avoid wrong interpretation/calculations.
- Usually layer modules are written so as to automate these conversions in Operating systems.

### How are macros different from inline functions?

- Macros are normally used whenever a set of instructions/tasks have to be repeatedly performed. They are small programs to carry out some predefined actions.

- We normally use the #define directive in case we need to define the values of some constants so in case a change is needed only the value can be changed and is reflected throughout.

```
#define mul(a,b) (a*b)
```

- The major disadvantage of macros is that they are not really functions and the usual error checking and stepping through of the code does not occur.

- Inline functions are expanded whenever it is invoked rather than the control going to the place where the function is defined and avoids all the activities such as saving the return address when a jump is performed. Saves time in case of short codes.

```
inline float add(float a,float b)
{
 return a+b;
}
```

- Inline is just a request to the compiler and it is upto to the compiler whether to substitute the code at the place of invocation or perform a jump based on its performance algorithms.

**What could be the reasons for a System to have gone blank and how would you Debug it?**

**Possible reasons could be:**

- PC being overheated.
- Dust having being accumulated all around.
- CPU fans not working properly .
- Faulty power connections.
- Faulty circuit board from where the power is being drawn.
- Support Drivers not having being installed.

**Debugging steps which can be taken are:**

- Cleaning the system thoroughly and maintaining it in a dust-free environment. Environment that is cool

enough and facilitates for easy passage of air should be ideal enough.

- By locating the appropriate support drivers for the system in consideration and having them installed.

### Explain interrupt latency and how can we decrease it?

1. Interrupt latency basically refers to the time span an interrupt is generated and it being serviced by an appropriate routine defined, usually the interrupt handler.
2. External signals, some condition in the program or by the occurrence of some event, these could be the reasons for generation of an interrupt.
3. Interrupts can also be masked so as to ignore them even if an event occurs for which a routine has to be executed.
4. Following steps could be followed to reduce the latency
  - ISRs being simple and short.
  - Interrupts being serviced immediately
  - Avoiding those instructions that increase the latency period.
  - Also by prioritizing interrupts over threads.
  - Avoiding use of inappropriate APIs.

### How to create a child process in linux?

- Prototype of the function used to create a child process is pid\_t fork(void);
- Fork is the system call that is used to create a child process. It takes no arguments and returns a value of type pid\_t.
- If the function succeeds it returns the pid of the child process created to its parent and child receives a zero value indicating its successful creation.
- On failure, a -1 will be returned in the parent's context, no child process will be created, and errno will be set.
- The child process normally performs all its operations in its parents context but each process independently of one another and also inherits some of the important attributes from it such as UID, current directory, root directory and so on.

### Significance of watchdog timer in Embedded Systems.

- Watchdog timer is basically a timing device that is set for predefined time interval and some event should occur during that time interval else the device generates a time out signal.
- One application where it is most widely used is when the mobile phone hangs and no activity takes place, in those cases watchdog timer performs a restart of the system and comes to the rescue of the users.
- It is used to reset to the original state whenever some inappropriate events take place such as too many commands being given at the same time or other activities that result in malfunctioning of the GUI. It is usually operated by counter devices.

### If you buy some RTOS, what are the features you look for in?

- Deterministic operating system having guaranteed worst-case interrupt latency and context-switch times.
- Documentation providing for the minimum, average, and maximum number of clock cycles required by each system call.
- Interrupt response times should be very minute.
- Context switch time should be very low.
- Compatibility with several plugin devices.
- Overall it should be very reliable.

### Why is java mostly used in embedded systems?

- Java was mainly designed and conceptualised for code that can work on different platforms without any hassles and also for being secure enough so as to not harm or corrupt other modules of code.
- Features like exception handling, simple syntax and Automatic Garbage collection all work in its favour as the language for use in ES's.
- Also that it is widely used in the form of Java applets makes it very popular confining it to the limits of JVM. It is Dynamic in nature.
- Its use is also being exploited in enterprise systems in the form of J2EE, J2SE, J2ME in case of mobile applications.

### Differentiate between mutexes vs semaphores.

- Semaphores is a synchronization tool to overcome the critical section problem.
- A semaphore S is basically an integer variable that apart from initialization is accessed only through atomic operations such as wait() and signal().
- Semaphore object basically acts as a counter to monitor the number of threads accessing a resource.
- Mutex is also a tool that is used to provide deadlock free mutual exclusion. It protects access to every critical data item. If the data is locked and is in use, it either waits for the thread to finish or awakened to release the lock from its inactive state.

### What are the commonly found errors in Embedded Systems?

- Damage of memory devices due to transient current and static discharges.
- Malfunctioning of address lines due to a short in the circuit.
- Malfunctioning of Data lines.
- Some memory locations being inaccessible in storage due to garbage or errors.
- Improper insertion of Memory devices into the memory slots.
- Faulty control signals.

### What is the need for having multibyte data input and output buffers in case of device ports?

- It's normally the case that some devices transfer the output either in a bursty or a sequential manner and also during input entry. If we take the example of keyboards, all the data entered is stored in a buffer and given at a time or one character at a time.
- In case of networking there may be several requests to access the same resource and all these are queued in a buffer and serviced in the order they are received. Hence to avoid the input/output units from getting overloaded with requests, we use multibyte buffers.

## What is the difference between Hardware design and Software Design?

- Hardware design is designed with the collaboration of interconnected parallel components that inherits the properties of each other. Whereas, Software design is designed with sequential components, that are based on objects and threads.
- Hardware design structure doesn't change dynamically and it can't be created, modified or removed easily. Whereas, Software design structure can be changed dynamically and re-usability features, used to define the components. It also includes easy creation, modification and removal of the components from the software.
- Hardware design focuses on individual components that are represented using analytical model that uses the transfer functions. Whereas, Software design represent the components using computation model that can have abstract execution engine or it can use the virtual machine that are non-deterministic.

## What are the differences between analytical and computational modeling?

- Analytical model allows the components to deal with the concurrency that are given during the process and the quantitative constraints that might come in between the components. Whereas, computational model deal with the non-deterministic abstraction hierarchy that has computational complexity to deal with the concurrency and allow it put also the physical constraints.
- Analytical models can't deal with the partial and incremental specifications that are non-deterministic. It is also not good in controlling the computation complexity that is used in the hardware design. Whereas, Computational model can, deal with constraints easily and it provides an upgradeable solution.
- Analytical model is the equation based model that doesn't have the time-sharing and parallelism concepts. Whereas, time-sharing and parallelism is used, in the abstract method that provides the theories of complexity and the real time evaluation.

## What are the functional requirements that are used in the embedded systems?

Functional requirements specifies the discrete and the logic to provide the services, functionality, features and the implementation that is independent from the components that are getting used in it. These are used to represent the constraints that are in the form of physical and define the probability to specify the components discretely from each other. The functional requirements are given for the hardware as well that gives more performance and measures the physical resources that are present like clock frequency, latency, etc. Functional requirements allow the system and hardware machines to transfer the functions with the non-deterministic probability.

## Why is Model transformations used in the embedded system?

Model transformations involve multiple models that are used to define different views of a system. It provides different level of granularity that it doesn't use either the top-down approach or the bottom-up approach to implement the basic functionality of the system. It is used to integrate the library components used that involves the iteration of the model that needs to be constructed. It also involves the analysis of the model so that the process can be made automated by using the construction tools. The compilation made the progress by improving the code that is written in high level language and the code generator produce the code that is required for the machine language.

## What is interaction semantics used in embedded systems?

Interaction semantics allow the actions to be performed by the system components to allow it to get the global behavior. The interaction can be atomic or non-atomic dependent on the interaction between the components. These components can't be modified using the interference having the other interactions. Languages that are used, having buffered communication, and other languages, that include multi-threaded languages that use non-atomic interactions. There are two types of interactions that are used:

- Strong synchronization: allow the components to participate together and have strong bonding in between.
- Weakly synchronizing: are asymmetric that required the communication from both the objects.

## What are the different types of system involved in embedded system?

Embedded systems are used to give the response in real time. So, it consists of the real time systems that allow the correct information to be passed to get the correct responses. For example, it includes of the flight control system that produce the responses in real time and it always take the values also in real time. If any delay been caused by the system then it deals in the fatal error. The real time system includes the

system that provides the response on time with the small delay. Real time systems include of many other system such as:

**Hard Real-Time Systems** - These are the systems that serve the purpose of having constraints that are hard and it totally depends on the time to provide the response on time.

**Soft Real-Time Systems** - These systems serve the purpose of having few delays in giving up the responses that can tolerate small variations.

**Hybrid Real-Time Systems** - These systems includes the properties from both the systems and increases the performance.

## What are the different types of Buses used by the embedded systems?

The buses are used to pass the messages between different components of the system. There are buses existing as:

- Memory Bus: it is related to the processor that is connected to the memory (RAM) using the data bus. This bus includes the collection of wires that are in series and runs parallel to each other to send the data from memory to the processor and vice versa.
- Multiplexed Address/Data Bus: Multiplex data bus consists of the bus that can read and write in the memory but it decreases the performance due to the time consumed in reading and writing of the data in the memory.
- De-multiplexed Bus: these consists of two wires in the same bus, where one wire consists of the address that need to be passed and the other one consists of the data that need to be passed from one to another. This is a faster method compared to other.
- Input/Output bus: it uses the multiplexing techniques to multiplex the same bus input and output signals. This creates the problem of having the deadlock due to slow processing of it.

## What is the main function of Multiplexed Address/Data Bus?

The memory bus is used to carry the address and the data from the processor to the memory so that it can be easily accessed by the devices. These buses carry the value of the data that has to be passed for the proper functioning. The use of the technique “Time division multiplexing” is used that allow the reading and writing of the data to be done from the same bus line. This requires lots of time to be given to the bus so that it can complete the read and write operation of the data in the memory. This is very expensive process due to the data transfer technique that is used in between the processor and the memory. This also gives the concept of cache and provides algorithms to solve the problems occurring in read and writes operations.

## How does Input/Output bus functions?

Input and output devices or functions allow the user to interact with the external files. Input and output functions are used to transfer the load on the bus. It uses the multiplexes having the input and output signals that remain same. Input and output buses move at the slower rate or speed than the processor speed. This increases the problem of bottleneck or the deadlock due to poor performance. There is a possibility to send more transistors for a layout to be given. These different devices may have very different speeds of communication. When programming IO bus control, make sure to take this into account.

In some systems, memory mapped IO is used. In this scheme, the hardware reads its IO from predefined memory addresses instead of over a special bus. This means you'll have simpler software, but it also means main memory will get more access requests.

## What is the function of simple thread poll in embedded system?

Simple thread poll allow the ready output to be passed for checking by giving it to the bus that is free and then the output is sent along the thread. The bus can send the output depending on the time that has been given and during the transfer the user won't be able to perform any other operation. The input is given after finding out the bus is free or not and if it free then a check is made to see that the input exists or not. This thread poll is easy to understand but it is not efficient method to allow the data to be put over the bus manually. The problem of not doing multi-tasking can occur due to doing one task at a time. The method is only be used when input/output occurs at interval that are infrequent.

## Why is it better to use multi-threading polling then single threading model?

Multi-threading allows a simple thread to be stored and polled. There is no Input/output function that is applied when it is having the poll. When there is no poll available to spawn it makes the system to sleep for an amount of time till the request for another poll reaches. If there is one process that is running then it divides that process into multiple threads and processes it accordingly. It allows the main thread to process all the request and produce the output by combining all other. Multi-threading allows the main thread not to put off the result or the output that will be generated. It also allow the priority of the thread to be changed by allowing to set the priority of the input/output process. It also has some problems with the polling interval that can make a thread starve for some time if the request isn't handled properly.

## How does the interrupt architecture works?

Interrupt architecture allows the use of interrupt by the processor whenever an Input/output is ready for the processing. The processor in this case calls a special function to handle the request that comes and leave all the work that is getting performed at that time. The special function that is known as interrupt handler or the interrupt service routine consists of all the input, and output queries, or the interrupts handled by it. It is an efficient and simple way to handle the interrupts. It uses only one function to deal with the interrupts. There are properties of starvation that can creep in when handling the input/output requests. The data can be lost if the interrupt doesn't get handled before the time runs out. This is a technique that is use to deal with the short processes that involve input and output.

## How does the interrupts handle by using the threads?

The interrupts that comes in between the input/output operations gets detected when the input/output devices are ready. The interrupt never gets handled directly rather, it sends the interrupt signal to the thread to the input/output device that is ready to allow the thread to take necessary actions. The thread uses the signaling concept that allows the initialization to be done using the semaphore that keeps the states updated and handle the interrupt in an easy way. The input/output device getting the request and it also passes the semaphore to handle. The input/output device takes the semaphore that is ready. The thread is having the minimum latency that uses the first level interrupt handler to handle the interrupts completely. It allows the priority of the thread to be set and it doesn't allow the context to be change as well.

## What is the function of DMA controlled in embedded system?

DMA stands for Direct Memory Access controller that handles the allocation of the memory dynamically to the components and allows the data to be transferred between the devices. It is used for the communication between different input/output devices. It automatically detects the devices that are present for the transfer of data between the input/output devices. The interrupt can be used to complete the data transfer between the devices. It is used to give the high quality performance as, the input/output device can perform the operations that are in parallel with the code that are in execution phase. It can't be used in all the systems. It consists of 8-bit microcontrollers that are used to control the overall system in execution.

## What are the different types of customizations that is used with the “volatile” keyword?

Volatile keyword is used to show that the value can be changed anytime in the program. It is used for the compiler purpose and for the customization that works with the normal variables that are stored in the memory. There are three types of optimizations associated with the “volatile” keyword:

- "Read" optimizations: allow the variable to be read once and put it in the register. If it is done then there is no re-reading of the variable during each and every time the program is compiled. The value can be used from the cache that is present in the register.
- "Write" optimizations: allow the variable to be written such that the last write of the variable will be considered and it will be processed on. This takes the normal values that are stored in the memory.
- Instruction reordering: allow to reorder the instructions that are used by the compiler and if any modification are required after being written once. The registers are used to perform the task and keep everything together.

## Write a program to show the functionality of Power-save super loop.

To check the loop time of the program the power-save super loop is used. If the average loop time of the program is 1ms, and it requires only few instructions to be checked every second the program will save the state and build a delay that will be caused to read the input on every loop and it saves lot of energy or the power that needs to be used. The function that is required to be performed to show the functionality is:

Main\_Function()

Function

```
{
 Initialization();
 Do_Forever
 {
 Check_Status();
 Do_Calculations();
 Output_Response();
 Delay_For_Next_Loop();
 }
}
```

## Why does pre-emptive multi-threading used to solve the central controller problem?

Multi-threading provide lot of functionality to the system to allow more than one task can be run at a time. It allows a process to execute faster with less difficulty. But, if there any problem comes in any program or the process than the entire system comes to a halt and slows down the whole system. To control the behavior of this the preemptive multi-threading is used. The control in this case is being shifted from one process to another at any time according to the requirement provided. It allows the program to give the control to another program that is having the higher priority. It includes of many problems like giving of a control by a process half way through in execution and the preemption of the process takes place then the data will be entered as corrupted in the memory location, multi-threading keeps the synchronization that is to be performed between different components of the system and the program and try to avoid the problem mentioned above.

## What are the rules followed by Mutexes?

Mutex is also called as Mutual Exclusion is a mechanism that is used to show the preemptive environment and allow providing security methods like preventing an unauthorized access to the resources that are getting used in the system. There are several rules that has to be followed to ensure the security policies:

- Mutex are directly managed by the system kernel that provides a secure environment to allow only the applications that passes the security rules and regulations. The mutex consists of objects that are allowed to be called by the kernel.

- Mutex can have only one process at a time in its area that is owned by the process using it. This allows less conflict between the different applications or processes that wait for their turn to execute it in the kernel area.
- Mutex can be allocated to another mutex that is running some task at a particular time and allow the kernel to have synchronization in between them.
- If Mutex is allocated to some other process then the area will consist of the process till the area is having the process in it.

## What is the purpose of using critical sections?

Critical section allows the process to run in an area that is defined by that process only. It is a sequence of instructions that can be corrupted if any other process tries to interrupt it. This process allow the operating system to give the synchronization objects that are used to monitor the processes that are up and running so that no other process will get executed till the critical region consists of a process that is already running. The example includes removal of the data from a queue running in a critical section and if not protected then it can be interrupted and the data have chances of getting corrupted. The processes exit from the critical section as soon as they finish the execution so that the chances can be given to other processes that are waiting for their chance to come.

## What is the function of Watchdog timer in embedded system?

The embedded system should have a function that can allow the fixing the system if anything goes wrong with it. Watchdog timer is a tool that is used in embedded system and having a long-fuse that runs several seconds. Watchdog timer includes the automated timing control that count down the number from max to 0 and when the counter reaches the zero, this WDT reset the micro-controller that gets turned off when the timer was in initial phases. Watchdog require the user to put some value before it runs out of time and reset the whole process as this can harm the data and the system. Resetting by WDT can be done when the process is half complete. Watchdog timer have the counter that is used to watch the processes that are running and if there is any issue occurs then WDT itself times out. The resetting of the system will be done to always give it the best possible way to execute the process.

## What is read modify write technique?

- Read modify write is a technique used to access the ports.
- Here in a single instruction all the three actions are combined.
- Here initially the data is read from the port then modified .
- At last the value is written back on the port.
- This feature saves many lines of code and makes the process easier.

In which addressing mode is the DPTR register used?

- Data pointer register is used in the indexed addressing mode.
- It is used in accessing data from the look-up table entries stored in ROM.
- SYNTAX: MOVC A, @A+DPTR
- Here c means the code (shows data elements are stored in code space of ROM).
- The contents of A are added to 16 bit DPTR to form the 16 bit address of the needed data.

Which registers are used for register indirect addressing mode if data is on-chip?

- R0 and R1 are the only registers used for register indirect addressing mode.
- These registers are 8 bit wide.
- Their use is limited to accessing only internal RAM.
- When these registers hold addresses of RAM, they must be preceded by a @ sign.
- In absence of this sign it will use the contents of register than the contents of memory location pointed by the same register.

Of the 128-byte internal RAM how many bytes are bit addressable?

- Only 16 bytes of the 128 bytes of RAM are bit addressable.
- The bit addressable RAM locations are 20H to 2FH.
- They are addressed as 0 to 127 (decimal) or 00 to 7F.
- Also the internal RAM locations 20 to 2FH are both byte and bit addressable.
- These 16 bytes can be accessed by single bit instructions using only direct addressing mode.

Explain parallel address space.

- The two physically separate memories using the same addresses form the parallel address space.
- Here the two memories are accessed using different access modes.

- This parallel addressing is generally used in 8052(enhaned version of 8051) having extra 128 bytes of RAM with addresses 80 to FFH and memory is called as upper memory.
- This distinguishes it from lower 128 bytes 00to 7FH.
- To access lower bytes direct addressing mode is used and for higher bytes indirect addressing mode is used.

Which port in 8051 performs a dual role?

- The port 3 can be used as simple input /output port or provides signals like interrupts.
- P3.0 and p3.1 are used for serial communication.
- P3.2 and P3.3 are used as external interrupts.
- P3.4 and P3.5 are used for timers.
- P3.6 and P3.7 are used as read write signals of external memory.

Explain high-end embedded processor.

- The microcontroller & processor can be interchangeably used here.
- It forms a general purpose processor .
- High-end stands for system of greatest power.
- Here when a microcontroller cannot do a specific task the processor does the same with higher power
- Example: ADM 64

Which 8051 version is using UV-EPROM?

- Ultraviolet EPROM is used in the version 8751.
- Here it becomes easier to erase the data using uv rays.
- It takes around 20 minutes to erase the data.
- To overcome this short coming flash ROM versions of 8751 available.

Which 8051 version uses Flash ROM?

- Atmel Corporation is manufacturing flash ROM
- At 89C51 chip of atmel is using flash ROM.
- Here no Rom eraser is needed.

- Erasing process is done by the PROM burner itself.
- To avoid use of PROM burner the 8052 version support in-system programming.

In which registers the immediate values cannot be directly loaded?

- The general purpose registers named A,B, R0-R7 can not accept the immediate data.
- To indicate immediate value pound sign is used before it.
- These register accept 8 bit data preceding a pound sign.
- There are registers present in 8051 used to store data temporarily.

Explain the difference between statements.

MOV A, #17H -a

MOV A, 17H -b

- Statement a indicates immediate data is copied into the register A.
- # in statement a indicates 17 is an immediate data which is moved to the destination.
- Here one copies the immediate data and other copies the data present at the specified address.
- In statement b the value present in 17H memory location is copied into A register.
- Absence of pound sign does not cause any error but required operation fails.

Which register is considered as the destination register for any arithmetic operation.

The 'A' register called the accumulator is used as the destination register.

**Example:**

ADD R2, # 12H

This is invalid as 12 is to be added to value present in R2 and data is also stored in R2.

ADD A,# 12H

This is valid as 12 is to be added to the value present in accumulator. Foregoing discussion explains why register A is used as accumulator.

Explain LCALL.

- It is called long call and is a 3 byte instruction.
- Here 1st byte is used for opcode & 2nd & 3rd bytes are used for address of target subroutine.
- Call subroutine can be used anywhere within 64 K- byte address space.

- Processor automatically saves on stack the address of instruction immediately below LCALL.
- After execution instruction return (RET) transfers control back to next instruction.

## How much time is required by an instruction for execution in 8051?

- Time required depends on the number of clock cycles used to execute instruction.
- These clock cycle are called machine cycle.
- Length of machine cycle of an instruction depends on frequency of crystal oscillator of controller.
- Also one machine cycle lasts 12 oscillator period so machine cycle is 1/12 of crystal frequency.
- Crystal frequency 8051 = 11.0592MHz
- Machine cycle =  $11.0592\text{MHz}/12 = 921.6\text{KHz} = 1.085\text{micro second}$ .

- [Next Page »](#)

## What is lst file?

- This file is also called as list file.
- It lists the opcodes ,addresses and errors detected by the assembler.
- List file is produced only when indicated by the user.
- It can be accessed by an editor and displayed on monitor screen or printed.
- Programmer uses this file to find the syntax errors and later fix them.

## How is a program executed' bit by bit' or' byte by byte'?

### EXAMPLE:

| ADDRESS | OPCODE | PROGRAM     |
|---------|--------|-------------|
| 1 0000  |        | ORG 0H      |
| 2 0000  | 7D25   | MOV R5,#25H |
| 3 0002  | 7F34   | MOV R7,#34H |
| 4 0004  | 2D     | ADD A, R5   |
| 5 0005  |        | END         |

- A program is always executed byte by byte.

- Firstly, 1st opcode 7D is fetched from location 0000 and then the value 25 is fetched from 0001 .
- 25 is then placed in the register R5 and program counter is incremented to point 0002.
- On execution of opcode 7F, value 34 is copied to register R7.
- Then addition of contents of R5 and accumulater takes place.
- Here all the opcodes are 8 bit forming a byte.

### Explain DB.

- DB is called as define byte used as a directive in the assembler.
- It is used to define the 8 bit data in binary ,hexadecimal or decimal formats.
- It is the only directive that can be used to define ASCII strings larger than two characters.
- DB is also used to allocate memory in byte sized chunks.
- The assembler always converts the numbers into hexadecimal.

### What is EQU?

- EQU is the equate assmbler directive used to define a constant without occupying a memory location.
- It associates a constant value with data label .
- Whenever the label appears in the program ,constant value is subsituted for label.
- Advantage: The constant value occuring at various positions in a program can be changed at once using this directive.
- Syntax: label EQU constant value

### How are labels named in assembly language?

- Label name should be unique and must contain alphabetic letters in both uppercase and lowercase.
- 1st letter should always be an alphabetic letter.
- It can also use digits and special characters ?, @, \_, \$.
- Label should not be one of the reserved words in assembly language.
- These labels make the program much easier to read and maintain.

### Are all the bits of flag register used in 8051?

- The flag register also called as the program status word uses only 6 bits.
- The two unused bits are user defineable flags.
- Carry, auxillary carry, parity and overflow flags are the conditional flags used in it.
- PSW.1 is a user definable bit and PSW.5 can be used as general purpose bit.
- Rest all flags indicate some or the other condition of an arithmetic operation.

### Which bit of the flag register is set when output overflows to the sign bit?

- The 2nd bit of the flag register is set when output flows to the sign bit.
- This flag is also called as the overflow flag.
- Here the output of the signed number operation is too large to be accomodated in 7 bits.
- For signed numbers the MSB is used to indicate the whether the number is positive or negative.
- It is only used to detect errors in signed number operations.

### Which register bank is used if we use the following instructions.

SETB PSW.3      A  
SETB PSW.4      B

- Statement A sets 3rd bit of flag register.
- Statement B sets 4th bit of flag register.
- Therefore register bank 3 is initiated .
- It uses memory location 18H to 1FH.
- The register bank is also called as R3.

### Issues related to stack and bank 1.

- Bank 1 uses the same RAM space as the stack.
- Stack pointer is incremented or decremented according to the push or pop instruction.
- If the stack pointer is decremented it uses locations 7,6,5... which belongs to register bank 0.
- If a given program uses R1 then stack is provided new memory location.
- The push instruction may also take stack to location 0 i.e.it will run out of space.

### Explain JNC.

- It is a command used to jump if no carry occurs after an arithmetic operation.
- It is called as jump if no carry( conditional jump instruction).

- Here the carry flag bit in PSW register is used to make decision.
- The processor looks at the carry flag to see if it is raised or not.
- If carry flag is 0 ,CPU fetches instructions from the address of the label.

### Can port 0 be used as input output port?

- Yes, port 0 can be used as input output port.
- Port 0 is an open drain unlike ports 2,3,4.
- To use it as input or output the 10k ohm pull-up resistors are connected to it externally.
- To make port 0 as input port it must be programmed by writing 1 to all bits.

**Example:**

MOV A,#0FFH

MOV P0,A

### Which 2 ports combine to form the 16 bit address for external memory access?

- Port0 and port2 together form the 16 bit address for external memory.
- Port0 uses pins 32 to 39 of 8051 to give the lower address bits(AD0-AD7)
- Port2 uses pins 21 to 28 of 8051 to give the higher address bits(A8-A15)
- This 16 bit address is used to access external memory if attached.
- When connected to external memory they cannot be used as input output ports.

### Can single bit of a port be accessed in 8051?

- Yes, 8051 has the capability of accessing only single bit of a port.
- Here only single bit is accessed and rest are unaltered.

**Syntax:** "SETB X. Y".

- Here X is the port number and y is the desired bit.

**Example:** SETB P1.2

Here the second bit of port 1 is set to 1.

## Other than SETB ,CLR are there any single bit instructions?

- There are total 6 single-bit instructions.
- CPL bit : complement the bit (bit= NOT bit).
- JB bit,target: Jump to target if bit equal to 1.
- JNB bit,target: Jump to target if bit is equal to 0.
- JCB bit,target: Jump to target if bit is equal to 1 and then clear bit.

- [Next Page »](#)

VLSI interview questions and answers for freshers and experienced candidates. Also find VLSI online practice tests to fight written tests and certification exams on VLSI. In this section we have covered almost all VLSI questions that might be asked during an interview.

## What are the steps required to solve setup and Hold violations in VLSI?

There are few steps that has to be performed to solved the setup and hold violations in VLSI. The steps are as follows:

- The optimization and restructuring of the logic between the flops are carried way. This way the logics are combined and it helps in solving this problem.
- There is way to modify the flip-flops that offer lesser setup delay and provide faster services to setup a device.  
Modifying the launch-flop to have a better hold on the clock pin, which provides CK->Q that makes the launch-flop to be fast and helps in fixing the setup violations.
- The network of the clock can be modified to reduce the delay or slowing down of the clock that captures the action of the flip-flop.
- There can be added delay/buffer that allows less delay to the function that is used.

## What are the different ways in which antenna violation can be prevented?

Antenna violation occurs during the process of plasma etching in which the charges generating from one metal strip to another gets accumulated at a single place. The longer the strip the more the charges gets accumulated. The prevention can be done by following method:

- Creating a jogging the metal line, that consists of atleast one metal above the protected layer.
- There is a requirement to jog the metal that is above the metal getting the etching effect. This is due to the fact that if a metal gets the etching then the other metal gets disconnected if the prevention measures are not taken.
- There is a way to prevent it by adding the reverse Diodes at the gates that are used in the circuits.

### What is the function of tie-high and tie-low cells?

Tie-high and tie-low are used to connect the transistors of the gate by using either the power or the ground. The gates are connected using the power or ground then it can be turned off and on due to the power bounce from the ground. The cells are used to stop the bouncing and easy from of the current from one cell to another. These cells are required Vdd that connects to the tie-high cell as there is a power supply that is high and tie-low gets connected to Vss. This connection gets established and the transistors function properly without the need of any ground bounce occurring in any cell.

### What is the main function of metastability in VSDL?

Metastability is an unknown state that is given as neither one or zero. It is used in designing the system that violates the setup or hole time requirements. The setup time requirement need the data to be stable before the clock-edge and the hold time requires the data to be stable after the clock edge has passed. There are potential violation that can lead to setup and hold violations as well. The data that is produced in this is totally asynchronous and clocked synchronous. This provide a way to setup the state through which it can be known that the violations that are occurring in the system and a proper design can be provided by the use of several other functions.

### What are the steps involved in preventing the metastability?

Metastability is the unknown state and it prevents the violations using the following steps:

1. proper synchronizers are used that can be two stage or three stage whenever the data comes from the asynchronous domain. This helps in recovering the metastable state event.
2. The synchronizers are used in between cross-clocking domains. This reduces the metastability by removing the delay that is caused by the data element that are coming and taking time to get removed from the surface of metal.
3. Use of faster flip-flops that allow the transaction to be more faster and it removes the delay time between the one component to another component. It uses a narrower metastable window that makes the delay happen but faster flip-flops help in making the process faster and reduce the time delay as well.

## What are the different design constraints occur in the Synthesis phase?

The steps that are involved in which the design constraint occurs are:

1. first the creation of the clock with the frequency and the duty cycle gets created. This clock helps in maintaining the flow and synchronizing various devices that are used.
2. Define the transition time according the requirement on the input ports.
3. The load values are specified for the output ports that are mapped with the input ports.
4. Setting of the delay values for both the input and output ports. The delay includes the input and output delay.
5. Specify the case-settings to report the correct time that are matched with the specific paths.
6. The clock uncertainty values are setup and hold to show the violations that are occurring.

## What are the different types of skews used in VLSI?

There are three types of skew that are used in VLSI. The skew are used in clock to reduce the delay or to understand the process accordingly. The skew are as follows:

### Local skew:

This contain the difference between the launching flip-flop and the destination flip-flop. This defines a time path between the two.

**Global skew:**

Defines the difference between the earliest component reaching the flip flow and the latest arriving at the flip flow with the same clock domain. In this delays are not measured and the clock is provided the same.

**Useful skew:**

Defines the delay in capturing a flip flop paths that helps in setting up the environment with specific requirement for the launch and capture of the timing path. The hold requirement in this case has to be met for the design purpose.

## What are the changes that are provided to meet design power targets?

To meet the design power target there should be a process to design with Multi-VDD designs, this area requires high performance, and also the high VDD that requires low-performance. This is used to create the voltage group that allow the appropriate level-shifter to shift and placed in cross-voltage domains. There is a design with the multiple threshold voltages that require high performance when the  $V_t$  becomes low. This have lots of current leakage that makes the  $V_t$  cell to lower the performance. The reduction can be performed in the leakage power as the clock in this consume more power, so placing of an optimal clock controls the module and allow it to be given more power. Clock tree allow the switching to take place when the clock buffers are used by the clock gating cells and reduce the switching by the power reduction.

## What are the different measures that are required to achieve the design for better yield?

To achieve better yeild then there should be reduction in maufacturability flaws. The circuit perfomance has to be high that reduces the parametric yield. This reduction is due to process variations The measures that can be taken are:

- Creation of powerful runset files that consists of spacing and shorting rules. This also consists of all the permissions that has to be given to the user.
- Check the areas where the design is having lithographic issues, that consists of sharp cuts.
- Use of redundant vias to reduce the breakage of the current and the barrier.

- Optimal placing of the de-coupling capacitances can be done so that there is a reduction in power-surges.

## What is the difference between the mealy and moore state machine?

- Moore model consists of the machine that have an entry action and the output depends only on the state of the machine, whereas mealy model only uses Input Actions and the output depends on the state and also on the previous inputs that are provided during the program.
- Moore models are used to design the hardware systems, whereas both hardware and software systems can be designed using the mealy model.
- Mealy machine's output depend on the state and input, whereas the output of the moore machine depends only on the state as the program is written in the state only.
- Mealy machine is having the output by the combination of both input and the state and the change the state of state variables also have some delay when the change in the signal takes place, whereas in Moore machine doesn't have glitches and its ouput is dependent only on states not on the input signal level.

## What is the difference between Synchronous and Asynchronous reset?

- Synchronous reset is the logic that will synthesize to smaller flip-flops. In this the clock works as a filter providing the small reset glitches but the glitches occur on the active clock edge, whereas the asynchronous reset is also known as reset release or reset removal. The designer is responsible of added the reset to the data paths.
- The synchronous reset is used for all the types of design that are used to filter the logic glitches provided between the clocks. Whereas, the circuit can be reset with or without the clock present.
- Synchronous reset doesn't allow the synthesis tool to be used easily and it distinguishes the reset signal from other data signal. The release of the reset can occur only when the clock is having its initial period. If the release happens near the clock edge then the flip-flops can be metastable.

## What are the different design techniques required to create a Layout for Digital Circuits?

The different design techniques to create the Layout for digital circuits are as follows:

- Digital design consists of the standard cells and represent the height that is required for the layout. The layout depends on the size of the transistor. It also consists of the specification for Vdd and GND metal paths that has to be maintained uniformly.
- Use of metal in one direction only to apply the metal directly. The metal can be used and displayed in any direction.
- Placing of the substrate that place where it shows all the empty spaces of the layout where there is resistances.
- Use of fingered transistors allows the design to be more easy and it is easy to maintain a symmetry as well.

## Write a program to explain the comparator?

To make a comparator there is a requirement to use multiplexer that is having one input and many outputs. This allows the choosing of the maximum numbers that are required to design the comparator. The implementation of the 2 bit comparator can be done using the law of trigotomy that states that  $A > B$ ,  $A < B$ ,  $A = B$  (Law of trigotomy). The comparator can be implemented using: combinational logic circuits or multiplexers that uses the HDL language to write the schematic at RTL and gate level.

Behavioral model of comparator represented like:

```
module comp0 (y1,y2,y3,a,b);
input [1:0] a,b;
output y1,y2,y3;
wire y1,y2,y3;
assign y1= (a >b)? 1:0;
assign y2= (b >a)? 1:0;
assign y3= (a==b)? 1:0;
endmodule
```

## What is the function of chain reordering?

The optimization technique that is used makes it difficult for the chain ordering system to route due to the congestion caused by the placement of the cells. There are tool available that automate the reordering of the chain to reduce the congestion that is produced at the first stage. It increases the problem of the chain system and this also allow the overcoming of the buffers that have to be inserted into the scan path. The increase of the hold time in the chain reordering can cause great amount of delay. Chain reordering allows the cell to be come in the ordered format while using the different clock domains. It is used to reduce the time delay caused by random generation of the element and the placement of it.

## What are the steps involved in designing an optimal pad ring?

- To make the design for an optimal pad ring there is a requirement for the corner-pads that comes across all the corners of the pad-ring. It is used to give power continuity and keep the resistance low.
- It requires the pad ring that is to fulfil the power domains that is common for all the ground across all the domains.
- It requires the pad ring to contain simultaneous switching noise system that place the transfer cell pads in cross power domains for different pad length.
- Drive strength is been seen to check the current requirements and the timings to make the power pads.
- Choose a no-connection pad that is used to fill the pad-frame when there is no requirement for the inputs to be given. This consumes less power when there is no input given at a particular time.
- Checking of the oscillators pads take place that uses the synchronous circuits to make the clock data synchronize with the existing one.

## What is the function of enhancement mode transistor?

The enhancement mode transistors are also called as field effect transistors as they rely on the electric field to control the shape and conductivity of the channel. This consists of one type of charge carrier in a semiconductor material environment. This also uses the unipolar transistors to differentiate themselves with the single-carrier type operation transistors that consists of the bipolar junction transistor. The uses of field effect transistor is to physical implementation of the semiconductor materials that is compared with the bipolar transistors. It provides with the majority of the charge carrier devices. The devices that consists of active channels to make the charge carriers pass through. It consists of the concept of drain and the source.

## What is the purpose of having Depletion mode Device?

Depletion modes are used in MOSFET it is a device that remains ON at zero gate-source voltage. This device consists of load resistors that are used in the logic circuits. This types are used in N-type depletion-load devices that allow the threshold voltages to be taken and use of -3 V to +3V is done. The drain is more positive in this comparison of PMOS where the polarities gets reversed. The mode is usually determined by the sign of threshold voltage for N-type channel. Depletion mode is the positive one and used in many technologies to represent the actual logic circuit. It defines the logic family that is dependent on the silicon VLSI. This consists of pull-down switches and loads for pull-ups.

## What is the difference between NMOS and PMOS technologies?

- PMOS consists of metal oxide semiconductor that is made on the n-type substrates and consists of active careers named as holes. These holes are used for migration purpose of the charges between the p-type and the drain. Whereas, NMOS consists of the metal oxide semiconductor and they are made on p-type substrates. It consists of electrons as their carriers and migration happens between the n-type source and drain.
- On applying the high voltage on the logic gates NMOS will be conducted and will get activated, whereas PMOS require low voltage to be activated.
- NMOS are faster than PMOS as the carriers that NMOS uses are electrons that travels faster than holes. The speed is twice as fast as holes.
- PMOS are more immune to noise than NMOS.

## What is the difference between CMOS and Bipolar technologies?

- CMOS technology allows the power dissipation to be low and it gives more power output, whereas bipolar takes lots of power to run the system and the circuit require lots of power to get activated.
- CMOS technology provides high input impedance that is low drive current that allow more current to be flown in the circuit and keep the circuit in a good position, whereas it provides high drive current means more input impedance.
- CMOS technology provides scalable threshold voltage more in comparison to the Bipolar technology that provides low threshold voltage.
- CMOS technology provides high noise margin, packing density whereas Bipolar technology allows to have low noise margin so that to reduce the high values and give low packing density of the components.

## What are the different classification of the timing control?

There are different classification in which the timing control data is divided and they are:

1. Delay based timing control: this is based on timing control that allows to manage the component such that the delay can be notified and wherever it is required it can be given. The delays that are based on this are as:

- Regular delay control: that controls the delay on the regular basis.
- Intra-assignment delay control: that controls the internal delays.
- Zero delay control

2. Events based timing control: this is based on the events that are performed when an event happens or a trigger is set on an event that takes place. It includes

- Regular event control
- Named event control
- Event OR control

3. Level sensitive timing control: this is based on the levels that are given like 0 level or 1 level that

is being given or shown and the data is being modified according the levels that are being set. When a level changes the timing control also changes.

- [Next Page »](#)

## What are Arrays?

An array is a series of elements. These elements are of the same type. Each element can be individually accessed using an index. For e.g an array of integers. Array elements are stored one after another (contiguous) in the memory. An array can have more than one dimension. First element in an array starts with 0.

## Explain two-dimensional array.

An array with two dimensions is called as a two-dimensional array. It is also called as a matrix. In C, a two dimensional array is initialized as int arr[nb\_of\_rows] [nb\_of\_columns]. Hence, two dimensional arrays can be considered as a grid. An element in a two dimensional can be accessed by mentioning its row and column. If the array has 20 integer values, it will occupy 80 bytes in memory (assuming 4 bytes for an integer). All of the bytes are in consecutive memory locations, the first row occupying the first 20 bytes, the second the next 20, and so on.

## Define Array of pointers.

An array of pointers is an array consisting of pointers. Here, each pointer points to a row of the matrix or an element. **E.g**

```
char *array [] = {"a", "b"}.
```

This is an array of pointers to characters.

- [Next Page »](#)

- [Next Page »](#)

## Describe the steps to insert data into a singly linked list.

Steps to insert data into a singly linked list:-

### Insert in middle

Data: 1, 2, 3, 5

1. Locate the node after which a new node (say 4) needs to be inserted.(say after 3)
2. create the new node i.e. 4
3. Point the new node 4 to 5 (`new_node->next = node->next(5)`)
4. Point the node 3 to node 4 (`node->next = newnode`)

### **Insert in the beginning**

Data: 2, 3, 5

1. Locate the first node in the list (2)
2. Point the new node (say 1) to the first node. (`new_node->next=first`) Insert in the end

### **Insert in the end**

Data: 2, 3, 5

1. Locate the last node in the list (5)
2. Point the last node (5) to the new node (say 6). (`node->next=new_node`)

### **Explain how to reverse singly link list.**

Reverse a singly link list using iteration:-

1. First, set a pointer ( `*current` ) to point to the first node i.e. `current=base`
2. Move ahead until `current!=null` (till the end)
3. set another pointer ( `*next` ) to point to the next node i.e. `next=current->next`
4. store reference of `*next` in a temporary variable ( `*result` ) i.e `current->next=result`
5. swap the result value with current i.e. `result=current`
6. and now swap the current value with next. i.e. `current=next`

7. return result and repeat from step 2

A linked list can also be reversed using recursion which eliminates the use of a temporary variable.

**Define circular linked list.**

An array of pointers is an array consisting of pointers. Here, each pointer points to a row of the matrix or an element.

**E.g**

```
char *array [] = {"a", "b"}.
```

This is an array of pointers to characters.

**Define circular linked list.**

In a circular linked list the first and the last node are linked. This means that the last node points to the first node in the list. Circular linked list allow quick access to first and last record using a single pointer.

**Describe linked list using C++ with an example.**

A linked list is a chain of nodes. Each and every node has at least two nodes, one is the data and other is the link to the next node. These linked lists are known as single linked lists as they have only one pointer to point to the next node. If a linked list has two nodes, one is to point to the previous node and the other is to point to the next node, then the linked list is known as doubly linked list.

To use a linked list in C++ the following structure is to be declared:

```
typedef struct List
{
 long Data;
 List* Next;
 List ()
 {
 Next=NULL;
 Data=0;
 }
}
```

```
};
typedef List* ListPtr.
```

The following code snippet is used to add a node.

```
void SLLList::AddANode()
{
 ListPtr->Next = new List;
 ListPtr=ListPtr->Next;
}
```

The following code snippet is used to traverse the list

```
void showList(ListPtr listPtr)
{
 while(listPtr!=NULL)
 {
 cout>>listPtr->Data;
 }
 return temp;
}
```

- [Next Page »](#)
- [Next Page »](#)

## What is linear search?

Linear search is the simplest form of search. It searches for the element sequentially starting from the first element. This search has a disadvantage if the element is located at the end. Advantage lies in the simplicity of the search. Also it is most useful when the elements are arranged in a random order.

## What is binary search?

Binary search is most useful when the list is sorted. In binary search, element present in the middle of the list is determined. If the key (number to search) is smaller than the middle element, the binary search is done on the first half. If the key (number to search) is greater than the middle element, the

binary search is done on the second half (right). The first and the last half are again divided into two by determining the middle element.

## Explain the bubble sort algorithm.

Bubble sort algorithm is used for sorting a list. It makes use of a temporary variable for swapping. It compares two numbers at a time and swaps them if they are in wrong order. This process is repeated until no swapping is needed. The algorithm is very inefficient if the list is long.

**E.g.** List: - 7 4 5 3

1. 7 and 4 are compared
2. Since  $4 < 7$ , 4 is stored in a temporary variable.
3. the content of 7 is now stored in the variable which was holding 4
4. Now, the content of temporary variable and the variable previously holding 7 are swapped.

## What is quick sort?

Quick sort is one the fastest sorting algorithm used for sorting a list. A pivot point is chosen. Remaining elements are portioned or divided such that elements less than the pivot point are in left and those greater than the pivot are on the right. Now, the elements on the left and right can be recursively sorted by repeating the algorithm.

- [Next Page »](#)
- [Next Page »](#)

## Describe stack operation.

Stack is a data structure that follows Last in First out strategy.

### Stack Operations:-

Push – Pushes (inserts) the element in the stack. The location is specified by the pointer.

Pop – Pulls (removes) the element out of the stack. The location is specified by the pointer

Swap: - the two top most elements of the stack can be swapped

Peek: - Returns the top element on the stack but does not remove it from the stack

Rotate:- the topmost (n) items can be moved on the stack in a rotating fashion

A stack has a fixed location in the memory. When a data element is pushed in the stack, the pointer points to the current element

## Describe queue operation.

Queue is a data structure that follows First in First out strategy.

### Queue Operations:

Push – Inserts the element in the queue at the end.

Pop – removes the element out of the queue from the front

Size – Returns the size of the queue

Front – Returns the first element of the queue.

Empty – to find if the queue is empty.

## Discuss how to implement queue using stack.

A queue can be implemented by using 2 stacks:-

1. An element is inserted in the queue by pushing it into stack 1
2. An element is extracted from the queue by popping it from the stack 2
3. If the stack 2 is empty then all elements currently in stack 1 are transferred to stack 2 but in the reverse order
4. If the stack 2 is not empty just pop the value from stack 2.

## Explain stacks and queues in detail.

A stack is a data structure based on the principle Last In First Out. Stack is container to hold nodes and has two operations - push and pop. Push operation is to add nodes into the stack and pop operation is to delete nodes from the stack and returns the top most node.

A queue is a data structure based on the principle First in First Out. The nodes are kept in an order. A node is inserted from the rear of the queue and a node is deleted from the front. The first element inserted is the first element first to delete.

## Question - What are priority queues?

A priority queue is essentially a list of items in which each item has associated with it a priority. Items are inserted into a priority queue in any, arbitrary order. However, items are withdrawn from a priority queue in order of their priorities starting with the highest priority item first.

Priority queues are often used in the implementation of algorithms.

## Question - What is a circular singly linked list?

In a circular singly linked list, the last node of the list is made to point to the first node. This eases the traveling through the list.

- [Next Page »](#)
- [Next Page »](#)

## Describe Trees using C++ with an example.

Tree is a structure that is similar to linked list. A tree will have two nodes that point to the left part of the tree and the right part of the tree. These two nodes must be of the similar type.

The following code snippet describes the declaration of trees. The advantage of trees is that the data is placed in nodes in sorted order.

```
struct TreeNode
{
 int item; // The data in this node.
 TreeNode *left; // Pointer to the left subtree.
 TreeNode *right; // Pointer to the right subtree.
}
```

The following code snippet illustrates the display of tree data.

```
void showTree(TreeNode *root)
{
 if (root != NULL)
 { // (Otherwise, there's nothing to print.)
 showTree(root->left); // Print items in left sub tree.
 cout << root->item << " "; // Print the root item.
```

```
 showTree(root->right); // Print items in right sub tree.
}
} // end inorderPrint()
```

## What is a B tree?

A B-tree of order m (the maximum number of children for each node) is a tree which satisfies the following properties :

1. Every node has  $\leq m$  children.
2. Every node (except root and leaves) has  $\geq m/2$  children.
3. The root has at least 2 children.
4. All leaves appear in the same level, and carry no information.
5. A non-leaf node with k children contains  $k - 1$  keys

## What are splay trees?

A splay tree is a self-balancing binary search tree. In this, recently accessed elements are quick to access again

It is useful for implementing caches and garbage collection algorithms.

When we move left going down this path, its called a zig and when we move right, its a zag.

Following are the splaying steps. There are six different splaying steps.

1. Zig Rotation (Right Rotation)
2. Zag Rotation (Left Rotation)
3. Zig-Zag (Zig followed by Zag)
4. Zag-Zig (Zag followed by Zig)
5. Zig-Zig
6. Zag-Zag

## What are red-black trees?

A red-black tree is a type of self-balancing binary search tree.

In red-black trees, the leaf nodes are not relevant and do not contain data.

Red-black trees, like all binary search trees, allow efficient in-order traversal of elements. Each node has a color attribute, the value of which is either red or black.

**Characteristics:**

- The root and leaves are black
- Both children of every red node are black.
- Every simple path from a node to a descendant leaf contains the same number of black nodes, either counting or not counting the null black nodes

## What are threaded binary trees?

In a threaded binary tree, if a node 'A' has a right child 'B' then B's left pointer must be either a child, or a thread back to A.

In the case of a left child, that left child must also have a left child or a thread back to A, and so we can follow B's left children until we find a thread, pointing back to A.

This data structure is useful when stack memory is less and using this tree the traversal around the tree becomes faster.

## What is a B+ tree?

It is a dynamic, multilevel index, with maximum and minimum bounds on the number of keys in each index segment. all records are stored at the lowest level of the tree; only keys are stored in interior blocks.

## Describe Tree database. Explain its common uses.

A tree is a data structure which resembles a hierarchical tree structure. Every element in the structure is a node. Every node is linked with the next node, either to its left or to its right. Each node has zero or more child nodes. The length of the longest downward path to a leaf from that node is known as the height of the node and the length of the path to its root is known as the depth of a node.

**Common Uses:**

- To manipulate hierarchical data

- Makes the information search, called tree traversal, easier.
- To manipulate data that is in the form of sorted list.
- To give visual effects for digital images using as a work flow by compositing them.

What is binary tree? Explain its uses.

A binary tree is a tree structure, in which each node has only two child nodes. The first node is known as root node. The parent has two nodes namely left child and right child.

#### **Uses of binary tree:**

- To create sorting routine.
- Persisting data items for the purpose of fast lookup later.
- For inserting a new item faster

How do you find the depth of a binary tree?

The depth of a binary tree is found using the following process:

1. Send the root node of a binary tree to a function
2. If the tree node is null, then return false value.
3. Calculate the depth of the left tree; call it 'd1' by traversing every node. Increment the counter by 1, as the traversing progresses until it reaches the leaf node. These operations are done recursively.
4. Repeat the 3rd step for the left node. Name the counter variable 'd2'.
5. Find the maximum value between d1 and d2. Add 1 to the max value. Let us call it 'depth'.
6. The variable 'depth' is the depth of the binary tree.

Explain pre-order and in-order tree traversal.

A non-empty binary tree is traversed in 3 types, namely pre-order, in-order and post-order in a recursive fashion.

#### **Pre-order:**

Pre-order process is as follows:

- Visit the root node
- Traverse the left sub tree
- Traverse the right sub tree

**In-Order:**

In order process is as follows:

- Traverse the left sub tree
- Visit the root node
- Traverse the right sub tree

**What is a B+ tree? Explain its uses.**

B+ tree represents the way of insertion, retrieval and removal of the nodes in a sorted fashion. Every operation is identified by a 'key'. B+ tree has maximum and minimum bounds on the number of keys in each index segment, dynamically. All the records in a B+ tree are persisted at the last level, i.e., leaf level in the order of keys.

B+ tree is used to visit the nodes starting from the root to the left or / and right sub tree. Or starting from the first node of the leaf level. A bi directional tree traversal is possible in the B+ tree.

**Define threaded binary tree. Explain its common uses**

A threaded binary tree is structured in an order that, all right child pointers would normally be null and points to the 'in-order successor' of the node. Similarly, all the left child pointers would normally be null and points to the 'in-order predecessor' of the node.

**Uses of Threaded binary tree:**

- Traversal is faster than unthreaded binary trees
- More subtle, by enabling the determination of predecessor and successor nodes that starts from any node, in an efficient manner.
- No stack overload can be carried out with the threads.
- Accessibility of any node from any other node
- It is easy to implement to insertion and deletion from a threaded tree.

**Explain implementation of traversal of a binary tree.**

Binary tree traversal is a process of visiting each and every node of the tree. The two fundamental binary tree traversals are 'depth-first' and 'breadth-first'.

The depth-first traversal are classified into 3 types, namely, pre-order, in-order and post-order.

**Pre-order:** Pre-order traversal involves visiting the root node first, then traversing the left sub tree and finally the right sub tree.

**In-order:** In-order traversal involves visiting the left sub tree first, then visiting the root node and finally the right sub tree.

**Post-order:** Post-order traversal involves visiting the left sub tree first, then visiting the right sub tree and finally visiting the root node.

The breadth-first traversal is the ‘level-order traversal’. The level-order traversal does not follow the branches of the tree. The first-in first-out queue is needed to traversal in level-order traversal.

## Explain implementation of deletion from a binary tree.

To implement the deletion from a binary tree, there is a need to consider the possibilities of deleting the nodes.

They are:

- Node is a terminal node: In case the node is the left child node of its parent, then the left pointer of its parent is set to NULL. In all other cases, if the node is right child node of its parent, then the right pointer of its parent is set to NULL.

- Node has only one child: In this scenario, the appropriate pointer of its parent is set to child node.

- Node has two children: The predecessor is replaced by the node value, and then the predecessor of the node is deleted.

- [Next Page »](#)
- [Next Page »](#)

## What is Service-Oriented Architecture?

SOA is an IT architecture strategy for business solution (and infrastructure solution) delivery based on the concept of service-orientation.

It is a set of components which can be invoked, and whose interface descriptions can be published and discovered.

It aims at building systems that are extendible, flexible and fit with legacy systems.

It promotes the re-use of basic components called services.

- [Next Page »](#)

.....  
Explain the meaning of Kernel.

### Answer

The kernel is the essential center of a computer operating system, the core that provides basic services for all other parts of the operating system.

As a basic component of an operating system, a kernel provides the lowest-level abstraction layer for the resources.

The kernel's primary purpose is to manage the computer's resources and allow other programs to run and use the resources like the CPU, memory and the I/O devices in the computer.

The facilities provided by the kernel are :

Memory management

The kernel has full access to the system's memory and must allow processes to access safely this memory as they require it.

Device management

To perform useful functions, processes need access to the peripherals connected to the computer, which are controlled by the kernel through device drivers

System calls

To actually perform useful work, a process must be able to access the services provided by the kernel.

Types of Kernel:

Monolithic kernels

Every part which is to be accessed by most programs which cannot be put in a library is in the kernel space:

Device drivers

Scheduler

Memory handling

File systems

Network stacks

Microkernels

In Microkernels, parts which really require to be in a privileged mode are in kernel space:

-Inter-Process Communication,

-Basic scheduling

-Basic memory handling

-Basic I/O primitives

Due to this, some critical parts like below run in user space:

The complete scheduler

Memory handling

File systems  
Network stacks

*Operating system - Kernel - Jan 07, 2010 at 15:00 PM by Vidya Sagar*

*Explain the meaning of Kernel.*

Kernel is the core module of an operating system. It is the kernel that loads first and retains in main memory of the computer system. It provides all essential operations /services those are needed by applications. Kernel takes the responsibility of managing the memory, task, disk and process

*What is a command interpreter?*

**Answer**

The part of an Operating System that interprets commands and carries them out.

A command interpreter is the part of a computer operating system that understands and executes commands that are entered interactively by a human being or from a program. In some operating systems, the command interpreter is called the shell.

The BIOS is looking for the files needed to load in case of Windows is the Command.com. The required files are Command.com, IO.sys, and Msdos.sys to get Windows started. They reside in the Root of the C Drive.

*Operating system - What is a command interpreter? - Jan 07, 2010 at 15:00 PM by Vidya Sagar*

*What is a command interpreter?*

A command interpreter is a program which reads the instructions given by the user. It then translates these instructions into the context of the operating system followed by the execution. Command interpreter is also known as 'shell'

*Explain the basic functions of process management.*

**Answer**

The basic functions of the OS wrt the process management are :

Allocating resources to processes,  
enabling processes to share and exchange information,  
protecting the resources of each process from other processes and  
enabling synchronisation among processes.

*Operating system - functions of process management - Jan 07, 2010 at 15:00 PM by Vidya Sagar*

*Explain the basic functions of process management.*

A process is an integral part of operating system. The resources are allocated by the operating system to the processes. The functions are:

- Allocation and protection of resources
- Synchronization enabling for all processes
- Processes protection

What is a named pipe?

**Answer**

A connection used to transfer data between separate processes, usually on separate computers. Its a pipe that an application opens by name in order to write data into or read data from the pipe. They are placed in the /dev directory and are treated as special files. Using a named pipe facilitates interprocess communications.

*Operating system - What is a named pipe? - Jan 07, 2010 at 15:00 PM by Vidya Sagar*

*What is a named pipe?*

A named pipe is an extension of the concept ‘pipe’ in multitasking operating system. Inter process communication is implemented using a named pipe. A pipe / traditional pipe is unnamed. The reason is it persists as long as the process is executing. Where as a named pipe is system-persistent and exists more than a process running time. It can be removed if not required in future.

What is pre-emptive and non-preemptive scheduling?

**Answer**

Tasks are usually assigned with priorities. At times it is necessary to run a certain task that has a higher priority before another task although it is running. Therefore, the running task is interrupted for some time and resumed later when the priority task has finished its execution. This is called preemptive scheduling.

Eg: Round robin

In non-preemptive scheduling, a running task is executed till completion. It cannot be interrupted.

Eg First In First Out

*Operating system - pre-emptive and non-preemptive scheduling - Jan 07, 2010 at 15:00 PM by Vidya Sagar*

*What is pre-emptive and non-preemptive scheduling?*

Preemptive scheduling: The preemptive scheduling is prioritized. The highest priority process should always be the process that is currently utilized.

Non-Preemptive scheduling: When a process enters the state of running, the state of that process is not deleted from the scheduler until it finishes its service time.

What is a semaphore?

**Answer**

A semaphore is a variable. There are 2 types of semaphores:

Binary semaphores

Counting semaphores

Binary semaphores have 2 methods associated with it. (up, down / lock, unlock)

Binary semaphores can take only 2 values (0/1). They are used to acquire locks. When a resource is available, the process in charge set the semaphore to 1 else 0.

Counting Semaphore may have value to be greater than one, typically used to allocate resources from a pool of identical resources.

**Operating System mutex - May 06, 2009 at 18:10 PM by Vidya Sagar**

*What is difference between binary semaphore and mutex?*

The differences between binary semaphore and mutex are:

- Mutex is used exclusively for mutual exclusion. Both mutual exclusion and synchronization can be used by binary.
- A task that took mutex can only give mutex.
- From an ISR a mutex can not be given.
- Recursive taking of mutual exclusion semaphores is possible. This means that a task that holds before finally releasing a semaphore, can take the semaphore more than once.
- Options for making the task which takes as DELETE\_SAFE are provided by Mutex, which means the task deletion is not possible when holding the mutex.

*Explain the meaning of mutex.*

**Answer**

A mutex and the binary semaphore are essentially the same. Both can take values: 0 or 1. However, there is a significant difference between them that makes mutexes more efficient than binary semaphores.

A mutex can be unlocked only by the thread that locked it. Thus a mutex has an owner concept.

*What are the different types of memory?*

**Answer**

The types of memory in a computer system are:

- Cache Memory - This is a small amounts of memory used to speed up system performance.
- Main memory - This is the RAM (random access memory)
- Secondary memory - This is a magnetic storage that keeps applications and data available to be used, and may also serve as virtual memory depending upon the operating system

*What are the different types of memory?*

The memory types are:

**SIMM** - Single-Line Memory Modules: Used to store single row of chips which are soldered onto Printed Circuit Board.

**DIMM** – Dual-Line Memory Modules: Used to store two rows of chips which are soldered onto printed circuit board and enables to contain two times memory than SIMM

**DRAM** – Dynamic Random Access Memory: It holds data for short time period and will be refreshed periodically.

**SDRAM** - Static RAM – Holds data and refreshing does not required. It is faster than DRAM.

**Flash Memory:** A non volatile, rewritable and solid state memory which performs the functions of both RAM and hard disk combined. Data is retained in the memory, in case of power loss. It is ideal for printers, cellular phones, digital cameras, pagers.

**Shadow RAM:** Allows the moving of selected parts of BIOS code that is available in ROM to the faster RAM.

### *What is RTOS?*

A certain capability within a specified time constraint is guaranteed by an operating system called 'real time operating system'. For example, certain object availability for a robot when it is assembled is ensured by a real time operating system. For making an object available within a designated time, the operating system would terminate with a failure. This is called a 'hard' real time operating system. The assembly line will be continued for functioning, but the output of production might be lower as the objects appearance is failed in a designated time, which causes the robot to be temporarily unproductive. This is called 'soft' real time operating system. Some of the real time operating systems qualities can be evaluated by operating systems such as Microsoft's Windows 2000, IBM's OS/390 up to some extent. It means that, if an operating system does not qualify certain characteristics of the operating system enables to be considered as a solution to a certain real time application problem.

There is a requirement of real time operating system in small embedded systems which are bundled as part of micro devices. The requirements of real time operating systems are considered for some kernels. Since device drivers also needed for a suitable solution, a real time operating system is larger than just a kernel.

### *What is the difference between hard real-time and soft real-time OS?*

Critical task completion on time is guaranteed by a hard real time system. The tasks needed for delays in the system are to be bounded by retrieving the stored data at the time which takes the operating system to complete any request.

A critical task obtains a priority over other tasks and maintaining that priority until the completion of the task. This is performed by a soft real time system. The system kernel delays need to be bounded as in the case of hard real time system.

### *What type of scheduling is there in RTOS?*

The tasks of real time operating system have 3 states namely, 'running', 'ready', 'blocked'. Only one task per CPU is being performed at a given point of time. In systems that are simpler, the list is usually short, two or three tasks at the most.

The designing of scheduler is the real key. Usually to minimize the worst-case length of time spent in the scheduler's critical section, the data structure of the ready list in the scheduler is designed. This is done during the inhibition of preemption. All interrupts are disabled in certain cases. The data structure choice depends on the tasks on the ready list can perform at the maximum.

### *What is interrupt latency?*

The time between a device that generates an interrupt and the servicing of the device that generated the interrupt is known as interrupt latency. Many operating systems' devices are serviced soon after the interrupt handler of the device is executed. The effect of interrupt latency may be caused by the interrupt controllers, interrupt masking, and the methods that handle interrupts of an operating system

### *What is priority inheritance?*

Priority inversion problems are eliminated by using a method called priority inheritance. The process priority will be increased to the maximum priority of any process which waits for any resource which has a resource lock. This is the programming methodology of priority inheritance.

When one or more high priority jobs are blocked by a job, the original priority assignment is ignored and execution of critical section at the highest priority level of jobs it blocks is performed. The job returns to the original priority level soon after executing the critical section. This is the basic idea of priority inheritance protocol.

*What is spin lock?*

In a loop a thread waits simply ('spins') checks repeatedly until the lock becomes available. This type of lock is a spin lock. The lock is a kind of busy waiting, as the threads remains active by not performing a useful task. The spin locks are to release explicitly, although some locks are released automatically when the thread blocks.

*What is a compiler?*

A compiler is a program that converts the programming language code to a machine readable format. The code should meet the standards, syntax of the programming language else the compiler will throw an error.

**OS - What is a compiler? - Oct 29, 2008, 17:35 pm by Amit Satpute**

*What is a compiler?*

A compiler is a program that takes a source code as an input and converts it into an object code. During the compilation process the source code goes through lexical analysis, parsing and intermediate code generation which is then optimized to give final output as an object code.

*Explain loader and linker.*

A loader loads the programs into the main memory from the storage device. The OS transparently calls the loader when needed.

A linker links and combines objects generated by a compiler into a single executable. A linker is also responsible to link and combine all modules of a program if written separately.

**OS - Explain loader and linker - Oct 29, 2008, 17:35 pm by Amit Satpute**

*Explain loader and linker.*

A loader is a program used by an operating system to load programs from a secondary to main memory so as to be executed.

Usually large applications are written into small modules and are then compiled into object codes. A linker is a program that combines these object modules to form an executable.

*Explain the types of memory.*

RAM- Random Access Memory is a volatile form of memory. This means that the content written on RAM is wiped off when the power is turned OFF.

ROM – Read only memory like hard disks, tapes are ROM where the content once written is permanently written unless wiped off by the user..

*Define Thread.*

A thread is the smallest unit of execution. A single operating system can have multiple threads running thereby creating a multithreading environment. Each thread has a priority based on which the process is executed.

**OS - Define Thread. - Oct 29, 2008, 17:35 pm by Amit Satpute**

*Define Thread.*

Threads are small processes that form parts of a larger process. A thread is contained inside a process. Different threads in the same process share some resources.

*What are the advantages of using Thread?*

Thread can be executed simultaneously. Multithreading allows the system to run faster. Therefore, threads increase performance. Multiple threads can share a single address space, all open files, and other resources. Threads usually reduce complexity when the aim is to provide security.

**OS - What are the advantages of using Thread? - Oct 29, 2008, 17:35 pm by Amit Satpute**

*What are the advantages of using Thread?*

Some advantages of using threads are:

- A process switching takes a longer time than that by threads.
- They can execute in parallel on a multiprocessor.
- Threads can share address spaces.

*Compare Thread and process.*

A thread is associated with a process. A process can have multiple threads running. When an application is started, a process is invoked and the process owns the memory, resources, and threads of execution that are associated with a running instance of an executable program. The program runs until the thread is associated with that process. A thread is the smallest unit of execution that is intact to a process.

**OS - Compare Thread and process. - Oct 29, 2008, 17:35 pm by Amit Satpute**

*Compare Thread and process.*

### Threads

Share address space  
Have direct access to data segment of its process  
Can communicate with other threads of the same process  
Have no overhead  
If a main thread gets affected, other threads too can get affected

### Processes

Have own address space  
Have own copy of data segment of the parent process  
Processes must use IPC for communication within sibling processes  
Have considerable overhead  
Change in a parent process has no effect on the child processes.

*What is multiprogramming?*

In multiprogramming, several programs run simultaneously on a single processor. Although in reality there is no real simultaneous execution of different programs, it gives a sense that they are executing simultaneously due to the part-by-part execution of all the programs.

*What is Multiprocessing?*

In multiprocessing, one or more programs are run by more than one processors.

### What is an lvalue and rvalue in C

Lvalue or location value is an address that is stored into while rvalue is the "read" value, ie., value of a constant or the value stored at a location. In the assignment

```
a = 31;
```

we have the lvalue of a used on the left hand side and the integer constant value 31 assigned to this location.

In the assignment

- [C](#)
- [interview](#)
- [programming](#)
- [5580 comments](#)
- [Read more](#)

### Embedded C interview Questions for Embedded Systems Engineers

Q: Which is the best way to write Loops?

Q: Is Count Down\_to\_Zero Loop better than Count\_Up\_Loops?

A: Always, count Down to Zero loops are better.

This is because, at loop termination, comparison to Zero can be optimized by compiler. (Eg using SUBS)  
Else, At the end of the loop there is ADD and CMP.

Q: What is loop unrolling?

A: Small loops can be unrolled for higher performance, with the disadvantage of increased codesize. When a loop is unrolled, a loop counter needs to be updated less often and fewer branches are executed. If the loop iterates only a few times, it can be fully unrolled, so that the loop overhead completely disappears.

eg:

```
int countbit1(uint n)
{ int bits = 0;
while (n != 0)
{
if (n & 1) bits++;
n >= 1;
}
return bits;
}
```

```
int countbit2(uint n)
{ int bits = 0;
while (n != 0)
{
if (n & 1) bits++;
if (n & 2) bits++;
if (n & 4) bits++;
if (n & 8) bits++;
n >= 4;
}
return bits;
}
```

Q: How does, taking the address of local variable result in unoptimized code?

A: The most powerful optimization for compiler is register allocation. That is it operates the variable from register, than memory. Generally local variables are allocated in registers. However if we take the address of a local variable, compiler will not allocate the variable to register.

Q: How does global variable result in unoptimized code?

A: For the same reason as above, compiler will never put the global variable into register. So its bad.

## Interrupt Handling, Interview Question for Embedded Systems Engineers

Q: How is ISR handled in vxworks ?

A: VxWorks supplies interrupt routines which connect to C functions and pass arguments to the functions to be executed at interrupt level. To return from an interrupt, the connected function simply returns. A routine connected to an interrupt in this way is referred to as an interrupt service routine (ISR) or interrupt handler. When an interrupt occurs, the registers are saved, a stack for the arguments to be passed is set up, then the C function is called. On return from the ISR, stack and registers are restored.

Q: How do you connect the C Function to any interrupt?

A: Using intConnect(INUM\_TO\_IVEC(intNum), intHandler, argToHandler).

The first argument to this routine is the byte offset of the interrupt vector to connect to. The second argument is the interrupt handler and the last is any argument to this handler.

Q: Is the C Code directly vectored to Hardware interrupts using IntConnect? Explain?

A: No. Interrupts cannot actually vector directly to C functions. Instead, intConnect( ) builds a wrapper, a small amount of code, that saves the necessary registers, sets up a stack entry (either on a special interrupt stack, or on the current task's stack) with the argument to be passed, and calls the connected function. On return from the function it restores the registers and stack, and exits the interrupt;

Q: Does the ISR use the stack of interrupted Task? Explain?

A: It depends on system architecture.

## Semaphore/Mutex Interview Question for Embedded Systems Engineers

Q: What are the different types of semaphores in vxworks? Which is the fastest?

A: VxWorks supports three types of semaphores. Binary, mutual exclusion, and counting semaphores. Binary is the fastest semaphore.

Q: When will you use binary semaphore ?

A: Binary semaphores are used for basic task synchronization and communication.

Q: When will you use mutual exclusion semaphore?

A: Mutual exclusion semaphores are sophisticated binary semaphores that are used to address the issues relating to task priority inversion and semaphore deletion in a multitasking environment.

Q: When will you use Counting semaphore?

A: Counting semaphores maintain a count of the number of times a resource is given. This is useful when

an action is required for each event occurrence. For example if you have ten buffers, and multiple tasks can grab and release the buffers, then you want to limit the access to this buffer pool using a counting semaphore.

Q: What is the difference between Mutex and binary semaphore?

A: The differences are:

- 1) Mutex can be used only for mutual exclusion, while binary can be used of mutual exclusion as well as synchronisation.
- 2) Mutex can be given only by the task that took it.
- 3) Mutex cannot be given from an ISR.
- 4) Mutual-exclusion semaphores can be taken recursively. This means that the semaphore can be taken more than once by the task that holds it before finally being released.
- 5) Mutex provides options for making the task that took it as DELETE\_SAFE. This means, that the task cannot be deleted when it holds mutex.

## VxWorks Task, Interview Questions

Q: What is a task in VxWorks?

A: A task is an independent program with its own thread of execution and execution context. VxWorks uses a single common address space for all tasks thus avoiding virtual-to-physical memory mapping. Every task contains a structure called the task control block that is responsible for managing the task's context.

Q: How do task's manage the context of execution ?

A: Every task contains a structure called the task control block that is responsible for managing the task's context. A task's context includes

- program counter or thread of execution
- CPU registers
- Stack of dynamic variables and function calls
- Signal handlers
- IO assignments
- Kernel control structures etc.,

Q: What are the Different states of tasks?

A task has 4 states. Read, Pend, Delay, Suspended

Q: Explain the task State transition ?

A: A task can be created with taskInit() and then activated with taskActivate() routine or both these actions can be performed in a single step using taskSpawn(). Once a task is created it is set to the suspend state and suspended until it is activated, after which it is added to the ready queue to be picked up by the scheduler and run. A task may be suspended by either the debugging your task, or the occurrence an exception. The difference between the pend and suspend states is that a task pends when it is waiting for a resource. A task that is put to sleep is added to delay queue.

## VxWorks Interview Question for Embedded Systems Engineers

Q: What is a RealTime System ?

A: A Real-time system is defined as a system where the response time for an event is predictable and deterministic with minimal latency.

Q: How is RTOS different from a general operating system?

A:

TaskScheduling:

RTOS generally have priority-based preemptive scheduling, which allows high-priority threads to meet their deadlines consistently. Whereas In a GPOS, the scheduler typically uses a "fairness" policy to dispatch threads and processes onto the CPU. Such a policy enables the high overall throughput required by desktop and server applications, but offers no assurances that high-priority, time-critical threads will execute in preference to lower-priority threads.

Preemptive Kernel:

In RTOS, kernel operations are preemptible. And so the RTOS kernel must be simple and elegant as possible.

Mechanisms to Avoid Priority Inversion:

This occurs when a lower-priority thread can inadvertently prevent a higher-priority thread from accessing the CPU. VxWorks specifically has protection for this.

Q: What is a task in VxWorks?

## what is the difference between linked list and array?

Storage:

1) Arrays are statically allocated whereas linked list are dynamically allocated and linked.

Generally, if the number of allocations are known before hand, we use arrays. Otherwise linked list

Performance:

2) Array is a high performance way of storing a group of data because each element is laid out next to its neighbor in memory. This allows for very fast access because (a) the code can do a little math and jump quickly to any location in the array, and (b) the elements are all grouped together so they tend to be in memory at the same time (fewer page faults and cache misses).

3) If you want to index by something like a string, you could use a hash table. Lets say you have a list of players, and you want to pull an object up from each player's name. You take the string, run a hash function to convert that into a number that falls within the range of an array, and store it in that element.

Or, you can have a linked list faster by breaking it into subelements. For example, if you were storing words in a dictionary, you could have an array of linked lists with 26 entries for a-z. That way when you manipulate a list, you are manipulating a much smaller subset.

Side note:

1) Java provides `arraylist` (**java.util.ArrayList**) option to programmers. So whether ArrayList better than array ?

If the data has a known number of elements or small fixed size upper bound, or where efficiency in using primitive types is important, arrays are often the best choice. However, many data storage problems are not that simple, and ArrayList (or one of the other Collections classes) might be the right choice.

2) How ArrayList are created ?

## Embedded System Interview Questions. Part II

### x86 interview questions

1. What does microprocessor speed depend on?
2. Is the address bus unidirectional?
3. Is the data bus Bi-directional?
4. What is the difference between microprocessor and microcontroller?
5. Why does microprocessor contain ROM chips?

6. Difference between static and dynamic RAM?
7. What is interrupt?
8. What is cache memory?
9. What is stack?
10. Can ROM be used as stack?
11. What is NV-RAM?

### C Interview Questions

1. What is the difference between malloc() and calloc() ?
2. write C code for deleting an element from a linked list traversing a linked list efficient way of eliminating duplicates from an array
3. Can structures be passed to the functions by value?
4. Why cannot arrays be passed by values to functions?
5. Advantages and disadvantages of using macro and inline functions?
6. What happens when recursion functions are declared inline?
7. What is the Scope of static variables?
8. What is the output of printf("\nab\bcd\ref");

## Embedded Systems Interview Questions

In general the question would be of two type

1) Questions concentrating more on general skills as a software engineer

On C lang, algorithms, design skills, Tools used etc

2) Questions concentrating more on specific skills in real-time system design

On RTOS,multi-tasking issues (synchronization,exclusion, scheduling),device driver skills (interrupt handling, device addressing)

Typical questions (assuming VxWorks Experience):

- 1) What is an Interrupt Service Routine? What are the typical requirements for an ISR?
- 2) What are the advantages and disadvantages of cache memory in a real-time system that uses DMA?
- 3) What are the different types of timing requirements that real-time systems have?
- 4) What is priority inversion and how is it typically addressed?

- 5) How many levels of task priority are there in VxWorks? Explain the task priority settings in your system?
- 6) You finished writing a real-time system but find that the system performs much worse than expected. What should you do? (Answers should include: hardware verification [cache, SDRAM and device timing], compiler settings, task priority settings, profiling, bad coding, etc.)

eg:

Solving issues in signaling to devices (ie being able to look at a logic analyser and find the problem with interdevice communication), Solving issues with data throughput (ie minimizing bus activity and processor usage to move data through a system), Service times for various threads/processes and activities ...

- 7) Discuss issues with dynamic memory allocation in real-time systems.  
(If they don't mention the words "memory leak" or "thread-safe", they should probably be excused.)
- 8) "What is the difference between the three types of semaphores in vxWorks'mutual-exclusion','binary', and 'counting'?"

## Linked List

What is linked list? Refresh the basics

Common Questions on linked lists ?

- 1) What's the difference between a linked list and an array?
- 2) Implement a linked list. Why did you pick the method you did?
- 3) Implement an algorithm to sort a linked list. Why did you pick the method you did? Now do it in O(n) time.
- 4) Implement an algorithm to reverse a linked list. Now do it without recursion.
- 5) Find the middle of a linked list. Now do it while only going through the list once
- 6) Implement an algorithm to insert a node into a circular linked list without traversing it.
- 7) How would you print out the data in a binary tree, level by level, starting at the top?
- 8) Write code for reversing a linked list.
- 9 ) Reverse the words in a sentence, i.e. "My name is Chris" becomes "Chris is name My." Optimize for speed. Optimize for space.
- 10) Implement Insert and Delete for

- singly-linked linked list
  - sorted linked list
  - circular linked list
- 

*Define the scope of static variables.*

The scope of a static variable is local to the block in which the variable is defined. However, the value of the static variable persists between two function calls.

**C - scope of static variables - Jan 11, 2010 at 14:55 PM by Vidya Sagar**

*Define the scope of static variables.*

Static variables in C have the scopes;

1. Static global variables declared at the top level of the C source file have the scope that they can not be visible external to the source file. The scope is limited to that file.
2. Static local variables declared within a function or a block, also known as local static variables, have the scope that, they are visible only within the block or function like local variables. The values assigned by the functions into static local variables during the first call of the function will persist / present / available until the function is invoked again.

The static variables are available to the program, not only for the function / block. It has the scope within the current compile. When static variable is declared in a function, the value of the variable is preserved , so that successive calls to that function can use the latest updated value. The static variables are initialized at compile time and kept in the executable file itself. The life time extends across the complete run of the program.

Static local variables have local scope. The difference is, storage duration. The values put into the local static variables, will still be present, when the function is invoked next time.

*What are volatile variables?*

Volatile variables get special attention from the compiler. A variable declared with the volatile keyword may be modified externally from the declaring function.

If the keyword volatile is not used, the compiler optimization algorithms might consider this to be a case of infinite loop. Declaring a variable volatile indicates to a compiler that there could be external processes that could possibly alter the value of that variable.

e.g.:

A variable that might be concurrently modified by multiple threads may be declared volatile. Variables declared to be volatile will not be optimized by the compiler. Compiler must assume that their values can change at any time. However, operations on a volatile variable are still not guaranteed to be atomic.

### C - What are volatile variables? - Jan 11, 2010 at 12:44 PM by Vidya Sagar

*What are volatile variables?*

Volatile variables are like other variables except that the values might be changed at any given point of time only by 'some external resources'.

Ex:

```
volatile int number;
```

The value may be altered by some external factor, though if it does not appear to the left of the assignment statement. The compiler will keep track of these volatile variables

*Explain the meaning of "Segmentation violation".*

A segmentation violation usually indicates an attempt to access memory which doesn't even exist.

### C - meaning of "Segmentation violation" - Jan 11, 2010 at 12:15 PM by Vidya Sagar

*Explain the meaning of "Segmentation violation".*

Segmentation violation usually occurs at the time of a program's attempt for accessing memory location, which is not allowed to access. The following code should create segmentation violation.

```
main() {
char *hello = "Hello World";
*hello = 'h';
}
```

At the time of compiling the program, the string "hello world" is placed in the binary mark of the program which is read-only marked. When loading, the compiler places the string along with other constants in the read-only segment of the memory. While executing a variable \*hello is set to point the character 'h', is attempted to write the character 'h' into the memory, which cause the segmentation violation. And, compiler does not check for assigning read only locations at compile time.

*What does static variable mean in C?*

Static variable is available to a C application, throughout the life time. At the time of starting the program execution, static variables allocations takes place first. In a scenario where one variable is to be used by all the functions (which is accessed by main () function), then the variable need to be declared as static in a C program. The value of the variable is persisted between successive calls to functions. One more significant feature of static variable is that, the address of the variable can be passed to modules and functions which are not in the same C file.

*What are bitwise shift operators?*

The bitwise operators are used for shifting the bits of the first operand left or right. The number of shifts is specified by the second operator.

Expression << or >> number of shifts

Ex:

```
number<<3; /* number is an operand - shifts 3 bits towards left*/
number>>2; /* number is an operand – shifts 2 bits towards right*/
```

The variable number must be an integer value.

For leftward shifts, the right bits those are vacated are set to 0. For rightward shifts, the left bits those are vacated are filled with 0's based on the type of the first operand after conversion.

If the value of 'number' is 5, the first statement in the above example results 40 and stored in the variable 'number'.

If the value of 'number' is 5, the second statement in the above example results 0 (zero) and stored in the variable 'number'.

### **C - bitwise shift operators - August 06, 2008 at 13:10 PM by Amit Satpute**

#### *What are bitwise shift operators?*

<< - Bitwise Left-Shift

Bitwise Left-Shift is useful when to want to MULTIPLY an integer (not floating point numbers) by a power of 2.

Expression: a << b

This expression returns the value of a multiplied by 2 to the power of b.

>> - Bitwise Right-Shift

Bitwise Right-Shift does the opposite, and takes away bits on the right.

Expression: a >> b

This expression returns the value of a divided by 2 to the power of b.

#### *Explain the use of bit field.*

Packing of data in a structured format is allowed by using bit fields. When the memory is a premium, bit fields are extremely useful. For example:

- Picking multiple objects into a machine word : 1 bit flags can be compacted
- Reading external file formats : non-standard file formats could be read in, like 9 bit integers

This type of operations is supported in C language. This is achieved by putting ':bit length' after the variable. Example:

```
struct packed_struct {
 unsigned int var1:1;
 unsigned int var2:1;
 unsigned int var3:1;
 unsigned int var4:1;
 unsigned int var5:4;
 unsigned int funny_int:9;
} pack;
```

packed-struct has 6 members: four of 1 bit flags each, and 1 4 bit type and 1 9 bit funny\_int.

C packs the bit fields in the structure automatically, as compactly as possible, which provides the maximum length of the field is less than or equal to the integer word length the computer system.

The following points need to be noted while working with bit fields:

The conversion of bit fields is always integer type for computation  
Normal types and bit fields could be mixed / combined  
Unsigned definitions are important.

*What is the purpose of "register" keyword?*

It is used to make the computation faster.

The register keyword tells the compiler to store the variable onto the CPU register if space on the register is available. However, this is a very old technique. Today's processors are smart enough to assign the registers themselves and hence using the register keyword can actually slowdown the operations if the usage is incorrect.

**C - purpose of "register" keyword - Jan 11, 2010 at 12:55 PM by Vidya Sagar**

*What is the purpose of "register" keyword?*

The keyword 'register' instructs the compiler to persist the variable that is being declared , in a CPU register.

Ex: register int number;

The persistence of register variables in CPU register is to optimize the access. Even the optimization is turned off; the register variables are forced to store in CPU register.

*Explain the use of "auto" keyword.*

When a certain variable is declared with the keyword 'auto' and initialized to a certain value, then within the scope of the variable, it is reinitialized upon being called repeatedly.

**C - use of "auto" keyword - Jan 11, 2010 at 14:50 PM by Vidya Sagar**

*Explain the use of "auto" keyword.*

The keyword 'auto' is used extremely rare. A variable is set by default to auto. The declarations:

auto int number; and int number;

has the same meaning. The variables of type static, extern and register need to be explicitly declared, as their need is very specific. The variables other than the above three are implied to be of 'auto' variables. For better readability auto keyword can be used.

*Compare array with pointer.*

The following declarations are NOT the same:

```
char *p;
char a[20];
```

The first declaration allocates memory for a pointer; the second allocates memory for 20 characters.

**C - Compare array with pointer - Jan 11, 2010 at 18:10 PM by Vidya Sagar**

*Define pointer in C.*

A pointer is an address location of another variable. It is a value that designates the address or memory location of some other value (usually value of a variable). The value of a variable can be accessed by a pointer which points to that variable. To do so, the reference operator (&) is pre-appended to the variable that holds the value. A pointer can hold any data type, including functions.

*What is NULL pointer?*

A null pointer does not point to any object.

NULL and 0 are interchangeable in pointer contexts. Usage of NULL should be considered a gentle reminder that a pointer is involved.

It is only in pointer contexts that NULL and 0 are equivalent. NULL should not be used when another kind of 0 is required.

**C - What is NULL pointer? - Jan 11, 2010 at 18:10 PM by Vidya Sagar**

*What is NULL pointer?*

A pointer that points to a no valid location is known as null pointer. Null pointers are useful to indicate special cases. For example, no next node pointer in case of a linked list. An indication of errors that pointers returned from functions.

#####
#####

#####
#####

## C question answer

#####
#####

Local Variables are stored in Stack. Register variables are stored in Register. Global & static variables are stored in data segment. The memory created dynamically are stored in Heap And the C program instructions get stored in code segment and the extern variables also stored in data segment.

constants are often left in ROM

#####
#####

### **Static Variable-**

There are 3 main uses for the static.

1. If you declare within a function: It retains the value between function calls
2. If it is declared for a function name: By default function is extern..so it will be visible from other files if the function declaration is as static..it is invisible for the outer files
3. Static for global variables:

By default we can use the global variables from outside files If it is static global..that variable is limited to with in the file.

if static variable declared in class-

static variables are those variables whose values are shared among various instances of a class.

For e.g. if you have a static variable "x" in class "A" and you create two instances of A i.e. a1 and a2. In that case a1 and a2 will share the common variable. This means if a2 changes the value of x than this will be changed for a1 as well.

Class A

```
{
static int x;
}
```

A a1=new A();

A a2=new A();

a1.x=10;

then a2.x will also become 10.

\*\* Default initial value of static integral type variables are zero.

\*If declared a static variable or function globally then its visibility will be limited to the file in which it has been declared

#####

### **What are the differences between structures and arrays?**

Ans: Structure is a collection of heterogeneous data types but array is a collection of homogeneous data types.

#### **Array**

1-It is a collection of data items of same data type.

2-It has declaration only

3-.There is no keyword.

4- array name represent the address of the starting element.

#### **Structure**

1-It is a collection of data items of different data type.

2- It has declaration and definition

3- keyword struct is used

4-Structure name is known as tag it is the short hand notation of the declaration.

#####

### **Union-**

A **union** is a special data type available in C that enables you to store different data types in the same memory location. You can define a union with many members, but only one member can contain a value at any given time.

Unions provide an efficient way of using the same memory location for multi-purpose.

```
union Data
{
 int i;
 float f;
```

```
char str[20];
} data
```

Now a variable of **Data** type can store an integer, a floating-point number, or a string of characters. This means that a single variable ie. same memory location can be used to store multiple types of data. You can use any built-in or user defined data types inside a union based on your requirement.

The memory occupied by a union will be large enough to hold the largest member of the union. For example, in above example Data type will occupy 20 bytes of memory space because this is the maximum space which can be occupied by character string. Following is the example which will display total memory size occupied by the above union:

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

union Data
{
 int i;
 float f;
 char str[20];
};

int main()
{
 union Data data;

 data.i = 10;
 data.f = 220.5;
 strcpy(data.str, "C Programming");

 printf("data.i : %d\n", data.i);
 printf("data.f : %f\n", data.f);
 printf("data.str : %s\n", data.str);

 return 0;
}
```

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

```
data.i : 1917853763
data.f : 4122360580327794860452759994368.000000
data.str : C Programming
```

```
#####
#####
```

### What is a structure?

Ans: Structure constitutes a super data type which represents several different data types in a single unit. A structure can be initialized if it is static or global

#####

### **What are the differences between structures and union?**

Ans: A structure variable contains each of the named members, and its size is large enough to hold all the members. Structure elements are of same size.

A union contains one of the named members at a given time and is large enough to hold the largest member. Union element can be of different sizes.

#####

### **What are the differences between structures and arrays?**

Ans: Structure is a collection of heterogeneous data type but array is a collection of homogeneous data types.

#### **Array**

- 1-It is a collection of data items of same data type.
- 2-It has declaration only
- 3-.There is no keyword.
- 4- array name represent the address of the starting element.

#### **Structure**

- 1-It is a collection of data items of different data type.
- 2- It has declaration and definition
- 3- keyword struct is used
- 4-Structure name is known as tag it is the short hand notation of the declaration.

#####

### **In header files whether functions are declared or defined?**

Ans: Functions are declared within header file. That is function prototypes exist in a header file,not function bodies

#####

### **malloc, realloc, calloc and free**

Malloc- Allocates a block of size bytes of memory, returning a pointer to the beginning of the block.

```
/* allocate node */
```

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

**Calloc-**

Calloc takes two arguments Calloc(b,c) where b no of object and c size of object

```
void* calloc (size_t num, size_t size);
#####
#####
```

**What are the differences between malloc () and calloc ()?**

Ans: Malloc Calloc 1-Malloc takes one argument Malloc(a);where a number of bytes 2-memory allocated contains garbage values

1-Calloc takes two arguments Calloc(b,c) where b no of object and c size of object

2-It initializes the contains of block of memory to zerosMalloc takes one argument, memory allocated contains garbage values.

It allocates contiguous memory locations. Calloc takes two arguments, memory allocated contains all zeros, and the memory allocated is not contiguous.

```
#####
#####
```

**Realloc-**

```
void* realloc (void* ptr, size_t size);
```

Changes the size of the memory block pointed to by *ptr*.

*ptr*-Pointer to a memory block previously allocated with malloc, calloc or realloc.

Alternatively, this can be a *null pointer*, in which case a new block is allocated (as if malloc was called).

*size*

New size for the memory block, in bytes.

size\_t is an unsigned integral type.

A pointer to the reallocated memory block, which may be either the same as *ptr* or a new location.

There are several possible outcomes with *realloc()*:

- If sufficient space exists to expand the memory block pointed to by *ptr*, the additional memory is allocated and the function returns *ptr*.
- If sufficient space does not exist to expand the current block in its current location, a new block of the size for *size* is allocated, and existing data is copied from the old block to the beginning of the new block. The old block is freed, and the function returns a pointer to the new block.
- If the *ptr* argument is NULL, the function acts like *malloc()*, allocating a block of *size* bytes and returning a pointer to it.
- If the argument *size* is 0, the memory that *ptr* points to is freed, and the function returns NULL.
- If memory is insufficient for the reallocation (either expanding the old block or allocating a new one), the function returns NULL, and the original block is unchanged.

```
#####
#####
```

**Free( ) -**

```
void free (void* ptr);
```

**Deallocate memory block**

A block of memory previously allocated by a call to malloc, calloc or realloc is deallocated, making it available again for further allocations.

If *ptr* does not point to a block of memory allocated with the above functions, it causes *undefined behavior*.

```
#####
```

**--What are macros? What are its advantages and disadvantages?**

**Definition:** In C and C++, a Macro is a piece of text that is expanded by the preprocessor part of the compiler. This is used in to expand text before compiling.

```
#define VALUE 10
```

Everywhere that VALUE is used the number of 10 will be used instead.

```
int fred[VALUE];
```

This declares an array of 10 ints.

Macros are abbreviations for lengthy and frequently used statements. When a macro is called the entire code is substituted by a single line though the macro definition is of several lines.

The advantage of macro is that it reduces the time taken for control transfer as in case of function.

The disadvantage of it is here the entire code is substituted so the program becomes lengthy if a macro is called several times.

```
#####
```

**Difference between pass by reference and pass by value?**

**Ans:** Pass by reference passes a pointer to the value. This allows the callee to modify the variable directly. Pass by value gives a copy of the value to the callee. This allows the callee to modify the value without modifying the variable. (In other words, the callee simply cannot modify the variable, since it lacks a reference to it.)

```
#####
```

**Difference between arrays and linked list?**

```
#####
#####
```

**enumerations-** An *enumeration* is a data type consisting of a set of named values that represent integral constants, known as *enumeration constants*Eg:

```
enum grain { oats, wheat, barley, corn, rice };
/* 0 1 2 3 4 */

enum grain { oats=1, wheat, barley, corn, rice };
/* 1 2 3 4 5 */

enum grain { oats, wheat=10, barley, corn=20, rice };
/* 0 10 11 20 21 */
```

```
#####
#####
```

**\*\* What are register variables? What are the advantages of using register variables?**

**Ans:** If a variable is declared with a register storage class,it is known as register variable.The register variable is stored in the cpu register instead of main memory.Frequently used variables are declared as register variable as it's access time is faster.

Registers are faster than memory to access, so the variables which are most frequently used in a C program can be put in registers using *register* keyword. The keyword *register* hints to compiler that a given variable can be put in a register.

It's compiler's choice to put it in a register or not. Generally, compilers themselves do optimizations and put the variables in register.

1) If you use & operator with a register variable then compiler may give an error or warning (depending upon the compiler you are using), because when we say a variable is a register, it may be stored in a register instead of memory and accessing address of a register is invalid

```
int main()
{
 register int i = 10;

 int *a = &i;

 printf("%d", *a);

 getchar();

 return 0;
}
```

2) *register* keyword can be used with pointer variables. Obviously, a register can have address of a memory location. There would not be any problem with the below program.

```
int main()
```

```
{
int i = 10;
register int *a = &i;
printf("%d", *a);
getchar();
return 0;
}
```

3) Register is a storage class, and C doesn't allow multiple storage class specifiers for a variable. So, *register* can not be used with *static*. Try below program.

```
int main()
{
 int i = 10;
 register static int *a = &i;
 printf("%d", *a);
 getchar();
 return 0;
}
```

4) There is no limit on number of register variables in a C program, but the point is compiler may put some variables in register and some not.

```
#####
#####
```

### Typedef-

a new name given to the existing data type may be easier to understand the code.

```
typedef struct {
 int scruples;
 int drams;
 int grains;
} WEIGHT;
```

The structure WEIGHT can then be used in the following declarations:

```
WEIGHT chicken, cow, horse, whale;
```

```
#####
#####
```

### typedef vs #define

The typedef interpretation is performed by the compiler whereas #define statements are processed by the pre-processor

Read variable field width in a scanf() from danis ritchie.  
fgets() and gets()

#####

What is recursion?

#####

Differentiate between for loop and a while loop? What are it uses?

Ans: For executing a set of statements fixed number of times we use for loop while when the number of iterations to be performed is not known in advance we use while loop.

#####

Near,Far and huge pointer-

Near, far, and huge pointers aren't part of standard C; they are/were an extension put in by several vendors to deal with segmented memory architectures

All of the stuff in this answer is relevant only to the old 8086 and 80286 segmented memory model.

near: a 16 bit pointer that can address any byte in a 64k segment

far: a 32 bit pointer that contains a segment and an offset. Note that because segments can overlap, two different far pointers can point to the same address.

huge: a 32 bit pointer in which the segment is "normalised" so that no two far pointers point to the same address unless they have the same value.

#####

Normalize pointer- In 8086 programming (MS DOS), a far pointer is *normalized* if its offset part is between 0 and 15 (0xF).

#####

**Red-black trees**- are self-balancing, and so can insert, delete, and search in  $O(\log n)$  time. Other types of balanced trees (e.g. AVL trees) are often slower for insert and delete operations.

In addition, the code for red-black trees tends to be simpler.

Red Black Tree is a special type of self balancing binary search tree. This is used as Syntax Trees in major compilers and as implementations of Sorted Dictionary.

Properties-

In addition to the requirements imposed on a binary search trees, with red–black trees:<sup>[4]</sup>

1. A node is either red or black.
2. The root is black. (This rule is sometimes omitted. Since the root can always be changed from red to black, but not necessarily vice-versa, this rule has little effect on analysis.)
3. All leaves (NIL) are black. (All leaves are same color as the root.)
4. Both children of every red node are black.
5. Every simple path from a given node to any of its descendant leaves contains the same number of black nodes.

#####

BITWISE OPERATOR-

Question- Write a program in C to know whether it is power of 2 or not.

```
int isPowerOfTwo (int x)
{
 return ((x != 0) && !(x & (x - 1)));
}
```

#####

Write a program in C to know whether a number is odd number or even using bit wise

Here is the simple program to find odd or even no using Bit wise operators:::::

The Logic is:: If the given number is ODD then it ends with 1 (LSB) in Binary format i.e num = 3, binary representation is 011.

If the given number is EVEN then it ends with 0 (LSB) in Binary format i.e num = 4, binary representation is 100.

Program is::::

-----

```
main()
{
int num;

printf("Enter the Number\n");
scanf("%d",&num);

if(num&1)
printf("Entered Number is ODD\n");
else
printf("Entered Number is EVEN\n");
}
```

#####

**Question: You are given two numbers A and B. Write a program to count number of bits needed to be flipped to convert A to B.**

Solution:

1. Calculate XOR of A and B.  
a\_xor\_b = A ^ B
2. Count the set bits in the above calculated XOR result.  
countSetBits(a\_xor\_b)

XOR of two number will have set bits only at those places where A differs from B.

Example:

```
A = 1001001
B = 0010101
a_xor_b = 1011100
No of bits need to flipped = set bit count in a_xor_b i.e. 4
```

Count set bits in an integer

August 19, 2009

Write an efficient program to count number of 1s in binary representation of an integer.

1. Simple Method Loop through all bits in an integer, check if a bit is set and if it is then increment the set bit count. See below program.

```
/* Function to get no of set bits in binary
```

representation of passed binary no. \*/

```
int countSetBits(unsigned int n)
{
 unsigned int count = 0;
 while(n)
 {
 count += n & 1;
 n >= 1;
 }
 return count;
}
```

```
#####
```

### Understanding “register” keyword in C

February 20, 2010

Registers are faster than memory to access, so the variables which are most frequently used in a C program can be put in registers using register keyword. The keyword register hints to compiler that a given variable can be put in a register. It's compiler's choice to put it in a register or not. Generally, compilers themselves do optimizations and put the variables in register.

1) If you use & operator with a register variable then compiler may give an error or warning (depending upon the compiler you are using), because when we say a variable is a register, it may be stored in a register instead of memory and accessing address of a register is invalid. Try below program.

```
int main()
{
 register int i = 10;
 int *a = &i;
 printf("%d", *a);
 getchar();
 return 0;
}
```

```
}
```

2) register keyword can be used with pointer variables. Obviously, a register can have address of a memory location. There would not be any problem with the below program.

```
int main()
{
 int i = 10;
 register int *a = &i;
 printf("%d", *a);
 getchar();
 return 0;
}
```

3) Register is a storage class, and C doesn't allow multiple storage class specifiers for a variable. So, register can not be used with static . Try below program.

```
int main()
{
 int i = 10;
 register static int *a = &i;
 printf("%d", *a);
 getchar();
 return 0;
}
```

4) There is no limit on number of register variables in a C program, but the point is compiler may put some variables in register and some not.

Please write comments if you find anything incorrect in the above article or you want to share more information about register keyword.

```
#####
#####
```

The first declaration:

```
const char * myPtr
```

declares a pointer to a constant character. You cannot use this pointer to change the value being pointed to:

```
char char_A = 'A';

const char * myPtr = &char_A;

*myPtr = 'J'; // error - can't change value of *myPtr
```

The second declaration,

```
char * const myPtr
```

declares a constant pointer to a character. The location stored in the pointer cannot change. You cannot change where this pointer points:

```
char char_A = 'A';

char char_B = 'B';

char * const myPtr = &char_A;

myPtr = &char_B; // error - can't change address of myPtr
```

The third declares a pointer to a character where both the pointer value and the value being pointed at will not change.

constant pointer to constant

```
const int *const ptr;
```

Above declaration is constant pointer to constant variable which means we cannot change value pointed by pointer as well as we cannot point the pointer to other variable

```
#####
```

### Returned values of printf() and scanf()

For example, below program prints geeksforgeeks 13

```
int main()
{
 printf("%d", printf("%s", "geeksforgeeks"));
 getchar();
}
```

Irrespective of the string user enters, below program prints 1.

```
int main()
{
 char a[50];
 printf(" %d", scanf("%s", a));
 getchar();
}

#####
#
```

#### Que- Nth node from the end of a Linked List

Method 1 (Use length of linked list)

- 1) Calculate the length of Linked List. Let the length be len.
- 2) Print the  $(len - n + 1)$ th node from the begining of the Linked List.

Method 2 (Use two pointers)

Maintain two pointers – reference pointer and main pointer. Initialize both reference and main pointers to head. First move reference pointer to  $n$  nodes from head. Now move both pointers one by one until reference pointer reaches end. Now main pointer will point to nth node from the end. Return main pointer.

#### Que- Function to check if a singly linked list is palindrome

#### Que-Write a function to get the intersection point of two Linked Lists.

Method 1(Simply use two loops)

Use 2 nested for loops. Outer loop will be for each node of the 1st list and inner loop will be for 2nd list. In the inner loop, check if any of nodes of 2nd list is same as the current node of first linked list. Time complexity of this method will be  $O(mn)$  where  $m$  and  $n$  are the number of nodes in two lists.

Method 2 (Mark Visited Nodes)

This solution requires modifications to basic linked list data structure. Have a visited flag with each node. Traverse the first linked list and keep marking visited nodes. Now traverse second linked list, If you see a visited node again then there is an intersection point, return the intersecting node. This solution works in  $O(m+n)$  but requires additional information with each node. A variation of this solution that doesn't require modification to basic data structure can be implemented using hash. Traverse the first linked list and store the addresses of visited nodes in a hash. Now traverse the second linked list and if you see an address that already exists in hash then return the intersecting node.

Method 3(Make circle in first list)

Thanks to Saravanan Man for providing below solution.

1. Traverse the first linked list(count the elements) and make a circular linked list. (Remember last node so that we can break the circle later on).
2. Now view the problem as find the loop in the second linked list. So the problem is solved.
3. Since we already know the length of the loop(size of first linked list) we can traverse those many number of nodes in second list, and then start another pointer from the beginning of second list. we have to traverse until they are equal, and that is the required intersection point.
4. remove the circle from the linked list.

Method 4 (Traverse both lists and compare addresses of last nodes) This method is only to detect if there is an intersection point or not. (Thanks to NeoTheSaviour for suggesting this)

- 1) Traverse the list 1, store the last node address
- 2) Traverse the list 2, store the last node address.
- 3) If nodes stored in 1 and 2 are same then they are intersecting.

Time complexity of this method is O(m+n) and used Auxiliary space is O(1)

#####

### Bitwise Operators:

[http://en.wikipedia.org/wiki/Bitwise\\_operation](http://en.wikipedia.org/wiki/Bitwise_operation)

What purpose do the bitwise AND, OR, XOR and the Shift operators serve?

Write a C program to count the bits set (bit value 1 ) in an integer? Find out and compare different possibilities?

Check if the 20th bit of a 32 bit integer is on or off?

Write a program in C to know whether it is power of 2 or not.

Write a program in C to know whether a number is odd number or even using bit wiseoperator.

Write a functionsetbits(x,p,n,y)that returns x with then bits that begin at position p set to therightmost n bits of y, leaving the other bits unchanged.

You are given two numbers A and B. Write a program to count number of bits needed to be flipped to convert A to B.

How to reverse the bits in an interger?

How to reverse the odd bits of an integer?

How would you count the number of bits set in a floating point number?

#####

### Storage for Strings in C

March 3, 2010

In C, a string can be referred either using a character pointer or as a character array.

Strings as character arrays

```
char str[4] = "GfG"; /*One extra for string terminator*/
```

```
/* OR */
char str[4] = {'G', 'f', 'G', '\0'}; /* '\0' is string terminator */
```

When strings are declared as character arrays, they are stored like other types of arrays in C. For example, if str[] is an auto variable then string is stored in stack segment, if it's a global or static variable then stored in data segment, etc.

### Strings using character pointers

Using character pointer strings can be stored in two ways:

#### 1) Read only string in a shared segment.

When string value is directly assigned to a pointer, in most of the compilers, it's stored in a read only block (generally in data segment) that is shared among functions.

```
char *str = "GfG";
```

In the above line "GfG" is stored in a shared read only location, but pointer str is stored in a read-write memory. You can change str to point something else but cannot change value at present str. So this kind of string should only be used when we don't want to modify string at a later stage in program.

#### 2) Dynamically allocated in heap segment.

Strings are stored like other dynamically allocated things in C and can be shared among functions.

```
char *str;
int size = 4; /*one extra for '\0'*/
str = (char *)malloc(sizeof(char)*size);
*(str+0) = 'G';
*(str+1) = 'f';
*(str+2) = 'G';
*(str+3) = '\0';
```

Let us see some examples to better understand above ways to store strings.

#### Example 1 (Try to modify string)

The below program may crash (gives segmentation fault error) because the line \*(str+1) = 'n' tries to write a read only memory.

```
int main()
{
```

```

char *str;
str = "GfG"; /* Stored in read only part of data segment */
(str+1) = 'n'; / Problem: trying to modify read only memory */
getchar();
return 0;
}

```

Below program works perfectly fine as str[] is stored in writable stack segment.

```

int main()
{
 char str[] = "GfG"; /* Stored in stack segment like other auto variables */
 (str+1) = 'n'; / No problem: String is now GnG */
 getchar();
 return 0;
}

```

Below program also works perfectly fine as data at str is stored in writable heap segment.

```

int main()
{
 int size = 4;

 /* Stored in heap segment like other dynamically allocated things */
 char *str = (char *)malloc(sizeof(char)*size);
 *(str+0) = 'G';
 *(str+1) = 'f';
 *(str+2) = 'G';
 *(str+3) = '\0';
 (str+1) = 'n'; / No problem: String is now GnG */
 getchar();
 return 0;
}

```

Example 2 (Try to return string from a function)

The below program works perfectly fine as the string is stored in a shared segment and data stored remains there even after return of getString()

```

char *getString()
{

```

```
char *str = "GfG"; /* Stored in read only part of shared segment */

/* No problem: remains at address str after getString() returns*/
return str;
}

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

The below program also works perfectly fine as the string is stored in heap segment and data stored in heap segment persists even after return of getString()

```
char *getString()
{
 int size = 4;
 char *str = (char *)malloc(sizeof(char)*size); /*Stored in heap segment*/
 *(str+0) = 'G';
 *(str+1) = 'f';
 *(str+2) = 'G';
 *(str+3) = '\0';

 /* No problem: string remains at str after getString() returns */
 return str;
}

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

But, the below program may print some garbage data as string is stored in stack frame of function getString() and data may not be there after getString() returns.

```
char *getString()
{
```

```

char str[] = "GfG"; /* Stored in stack segment */

/* Problem: string may not be present after getString() returns */
return str;
}

int main()
{
 printf("%s", getString());
 getchar();
 return 0;
}

```

#####

### Evaluation order of operands

August 14, 2010

Consider the below C/C++ program.

```

#include<stdio.h>
int x = 0;

int f1()
{
 x = 5;
 return x;
}

int f2()
{
 x = 10;
 return x;
}

int main()
{
 int p = f1() + f2();
 printf("%d ", x);
 getchar();
 return 0;
}

```

What would the output of the above program - '5' or '10'?

The output is undefined as the order of evaluation of `f1() + f2()` is not mandated by standard. The compiler is free to first call either `f1()` or `f2()`. Only when equal level precedence operators appear in an expression, the associativity comes into picture. For example, `f1() + f2() + f3()` will be considered as `(f1() + f2()) + f3()`. But among first pair, which function (the

operand) evaluated first is not defined by the standard.

```
#####
#####
```

## Comma in C and C++

August 29, 2010

In C and C++, comma (,) can be used in two contexts:

- 1) Comma as an operator:

The comma operator (represented by the token ,) is a binary operator that evaluates its first operand and discards the result, it then evaluates the second operand and returns this value (and type). The comma operator has the lowest precedence of any C operator, and acts as a sequence point.

```
/* comma as an operator */
int i = (5, 10); /* 10 is assigned to i*/
int j = (f1(), f2()); /* f1() is called (evaluated) first followed by f2().
The returned value of f2() is assigned to j */
```

- 2) Comma as a separator:

Comma acts as a separator when used with function calls and definitions, function like macros, variable declarations, enum declarations, and similar constructs.

```
/* comma as a separator */
int a = 1, b = 2;
void fun(x, y);
```

The use of comma as a separator should not be confused with the use as an operator. For example, in below statement, f1() and f2() can be called in any order.

```
/* Comma acts as a separator here and doesn't enforce any sequence.
Therefore, either f1() or f2() can be called first */
void fun(f1(), f2());
See this for C vs C++ differences of using comma operator.
```

You can try below programs to check your understanding of comma in C.

```
// PROGRAM 1
#include<stdio.h>
int main()
{
 int x = 10;
 int y = 15;

 printf("%d", (x, y));
 getchar();
 return 0;
}
// PROGRAM 2: Thanks to Shekhu for suggesting this program
#include<stdio.h>
int main()
{
 int x = 10;
```

```

int y = (x++, ++x);
printf("%d", y);
getchar();
return 0;
}
// PROGRAM 3: Thanks to Venki for suggesting this program
int main()
{
 int x = 10, y;

 // The following is equivalent to y = x++
 y = (x++, printf("x = %d\n", x), ++x, printf("x = %d\n", x), x++);

 // Note that last expression is evaluated
 // but side effect is not updated to y
 printf("y = %d\n", y);
 printf("x = %d\n", x);

 return 0;
}

```

#####

## Operands for sizeof operator

February 9, 2010

In C, sizeof operator works on following kind of operands:

- 1) type-name: type-name must be specified in parentheses.

sizeof (type-name)

- 2) expression: expression can be specified with or without the parentheses.

sizeof expression

The expression is used only for getting the type of operand and not evaluated. For example, below code prints value of i as 5.

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

```

int main()
{
 int i = 5;
 int int_size = sizeof(i++);
 printf("\n size of i = %d", int_size);
 printf("\n Value of i = %d", i);

 getchar();
 return 0;
}
```

Output of the above program:

size of i = depends on compiler

value of i = 5

```
#####
```

### Result of comma operator as l-value in C and C++

January 19, 2011

Using result of comma operator as l-value is not valid in C. But in C++, result of comma operator can be used as l-value if the right operand of the comma operator is l-value.

For example, if we compile the following program as a C++ program, then it works and prints b = 30. And if we compile the same program as C program, then it gives warning/error in compilation (Warning in Dev C++ and error in Code Blocks).

```
#include<stdio.h>

int main()
{
 int a = 10, b = 20;
 (a, b) = 30; // Since b is l-value, this statement is valid in C++, but not in C.
 printf("b = %d", b);
 getchar();
 return 0;
}
C++ Output:
b = 30
```

```
#####
```

### What is evaluation order of function parameters in C?

June 11, 2010

It is compiler dependent in C. It is never safe to depend on the order of evaluation of side effects. For example, a function call like below may very well behave differently from one compiler to another:

```
void func (int, int);
```

```
int i = 2;
func (i++, i++);
```

There is no guarantee (in either the C or the C++ standard language definitions) that the increments will be evaluated in any particular order. Either increment might happen first. func might get the arguments '2, 3', or it might get '3, 2', or even '2, 2'.

```
#####
```

### Order of operands for logical operators

January 3, 2011

The order of operands of logical operators &&, || are important in C/C++.

In mathematics, logical AND, OR, EXOR, etc... operations are commutative. The result will not change even if we swap RHS and LHS of the operator.

In C/C++ (may be in other languages as well) even though these operators are commutative, their order is critical. For example see the following code,

```
// Traverse every alternative node
while(pTemp && pTemp->Next)
{
 // Jump over to next node
 pTemp = pTemp->Next->Next;
}
```

The first part pTemp will be evaluated against NULL and followed by pTemp->Next. If pTemp->Next is placed first, the pointer pTemp will be dereferenced and there will be runtime error when pTemp is NULL.

It is mandatory to follow the order. Infact, it helps in generating efficient code. When the pointer pTemp is NULL, the second part will not be evaluated since the outcome of AND (&&) expression is guaranteed to be 0.

```
#####
#####
```

### **Static functions in C**

May 5, 2010

In C, functions are global by default. The “static” keyword before a function name makes it static. For example, below function fun() is static.

```
static int fun(void)
{
 printf("I am a static function ");
}
```

Unlike global functions in C, access to static functions is restricted to the file where they are declared. Therefore, when we want to restrict access to functions, we make them static. Another reason for making functions static can be reuse of the same function name in other files.

For example, if we store following program in one file file1.c

```
/* Inside file1.c */
static void fun1(void)
{
 puts("fun1 called");
}
```

And store following program in another file file2.c

```
/* Inside file2.c */
int main(void)
{
 fun1();
 getchar();
 return 0;
}
```

Now, if we compile the above code with command “gcc file2.c file1.c”, we get the error “undefined reference to `fun1`”. This is because fun1() is declared static in file1.c and cannot be used in file2.c.

```
#####
#####
```

### **exit(), abort() and assert()**

July 9, 2010

`exit()`

`void exit ( int status );`

`exit()` terminates the process normally.

`status`: Status value returned to the parent process. Generally, a status value of 0 or `EXIT_SUCCESS` indicates success, and any other value or the constant `EXIT_FAILURE` is used to indicate an error. `exit()` performs following operations.

- \* Flushes unwritten buffered data.

- \* Closes all open files.

- \* Removes temporary files.

- \* Returns an integer exit status to the operating system.

The C standard `atexit()` function can be used to customize `exit()` to perform additional actions at program termination.

Example use of `exit()`.

```
/* exit example */
#include <stdio.h>
#include <stdlib.h>

int main ()
{
 FILE * pFile;
 pFile = fopen ("myfile.txt", "r");
 if (pFile == NULL)
 {
 printf ("Error opening file");
 exit (1);
 }
 else
 {
 /* file operations here */
 }
 return 0;
}
```

`abort()`

`void abort ( void );`

Unlike `exit()` function, `abort()` may not close files that are open. It may also not delete temporary files and may not flush stream buffer. Also, it does not call functions registered with `atexit()`.

This function actually terminates the process by raising a `SIGABRT` signal, and your program can include a handler to intercept this signal (see this).

So programs like below might not write “Geeks for Geeks” to “tempfile.txt”

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

```

int main()
{
 FILE *fp = fopen("C:\\myfile.txt", "w");

 if(fp == NULL)
 {
 printf("\n could not open file ");
 getchar();
 exit(1);
 }

 fprintf(fp, "%s", "Geeks for Geeks");

 /* */
 /* */
 /* Something went wrong so terminate here */
 abort();

 getchar();
 return 0;
}

```

If we want to make sure that data is written to files and/or buffers are flushed then we should either use exit() or include a signal handler for SIGABRT.

assert()

void assert( int expression );

If expression evaluates to 0 (false), then the expression, sourcecode filename, and line number are sent to the standard error, and then abort() function is called. If the identifier NDEBUG ("no debug") is defined with #define NDEBUG then the macro assert does nothing.

Common error outputting is in the form:

Assertion failed: expression, file filename, line line-number

#include<assert.h>

```

void open_record(char *record_name)
{
 assert(record_name != NULL);
 /* Rest of code */
}

int main(void)
{
 open_record(NULL);
}

#####

```

Dear All,

C Questions discussed on 22 April 2013 are listed below. I don't know upto what extend these solutions are correct. Please let me know if they are wrong

#### Q1 Traverse binary tree breadth-wise

Ans:

```
void print(BinaryTree *root)
{
if (!root) return;
queue<BinaryTree*> currentLevel, nextLevel;
currentLevel.push(root);
while (!currentLevel.empty())
{
BinaryTree *currNode = currentLevel.front();
currentLevel.pop();
if (currNode)
{
printf(currNode->data) ;
nextLevel.push(currNode->left);
nextLevel.push(currNode->right);
}
if (currentLevel.empty())
{
printf("end");
swap(currentLevel, nextLevel);
}
}
}
```

find out least common ancestor in tree

```
void find(root, node1 , node2)
{
if(root==null || node1 ==null || node2==null)
return null;
```

```

if(node1== root)
return node1;
if(node2==root)
return node2;
if(node1 < root && node2 < root)
{
return find(root->left,node1,node2);
}
elseif(node1 > root && node2 > root)
{
return find(root->right,node1,node2);
}
return root;
}

```

Dear All,

These are the C Questions discussed on 19-04-2013, listed below.

**Q1.) Reverse a string using bitwise operator?**

Ans: using xor operator.

```

Int main(){
Char x[] = "Manohar";
Char *string = x;
int len = strlen(string);
for (int i=0; i<len/2; i++){
x[len-i-1] ^= x[i];
x[i] ^= x[len-i-1];
x[len-i-1] ^= x[i];
}
}

```

**Q2.) printf("%d", -1 <<4); what is the output ?**

Ans:1 is represented in binary form is 0000 0000 0000 0001

-1 means 1's compliment                    1111 1111 1111 1110

Output is FFF0

**Q3.) Use of bit fields in C?**

Ans: Both C and C++ allow integer members to be stored into memory spaces smaller than the compiler would ordinarily allow. Width in bit fields explicitly declared and A bit field declaration may not use either of the type qualifiers, const or volatile.

**The following restrictions apply to bit fields. You cannot:**

- &#61623 Define an array of bit fields
- &#61623 Take the address of a bit field
- &#61623 Have a pointer to a bit field
- &#61623 Have a reference to a bit field

The following example demonstrates padding. The structure kitchen contains eight members totaling 16 bytes.

```

struct on_off {
unsigned light : 1;
unsigned toaster : 1;

```

```

int count; /* 4 bytes */
unsigned ac : 4;
unsigned : 4;
unsigned clock : 1;
unsigned : 0;
unsigned flag : 1;
} kitchen ;

```

| Member Name          | Storage Occupied                         |
|----------------------|------------------------------------------|
| light                | 1 bit                                    |
| toaster              | 1 bit                                    |
| (padding -- 30 bits) | To the next int boundary                 |
| count                | The size of an int (4 bytes)             |
| ac                   | 4 bits                                   |
| (unnamed field)      | 4 bits                                   |
| clock                | 1 bit                                    |
| (padding -- 23 bits) | To the next int boundary (unnamed field) |
| flag                 | 1 bit                                    |
| (padding -- 31 bits) | To the next int boundary                 |

Q4.) what is the output of given code ?

```

Int x = 5;
Printf("%d %d %d",x, x<<2, x>>2);

```

Ans: 5 20 1

Q5.) char \*ptr=NULL;

```
Char arr[10] = "abcd";
```

```
Strcpy(ptr,arr);
```

```
Printf("%s",ptr);
```

What is the output?

Ans: segmentation fault. Since pointer is not pointing to any valid memory.

Q6.) Int x[]={1,2,3,4,5};

```
Int *ptr, **ptr2;
```

```
Ptr = x;
```

```
Ptr2 = &ptr ;
```

How do you update x[2]= 10 using ptr2 ?

Ans: ptr2 + 2 and \*\*ptr = 10;

Or

```
**(ptr2 + 2) = 10;
```

Q7.) Convert the expression ((A + B) \* C - (D - E) ^ (F + G)) to postfix notation.

Ans: AB + C \* DE - - FG + ^

Q8.) Predict the output from the given code?

```

Struct abc{
int a;
float b;
}d;
main(){
d.a = 1;
printf("%f" , d.b);
}

```

```
}
```

Ans: 0.0

**Q9.)Predict the which one is not valid statement?**

- 1.++d+++e++;
- 2.f\*=g+=h=5;
- 3.L-- >> M << -- N;
- 4.int a(int a, b = 0, c =a((c=b,++b));
- 5.i- >j< -k

Ans: 5

Thank you!

Best Regards,

Manohar

**Q4:**

```
int p1 = sizeof("HELLO");
Int p2 = strlen("HELLO");
```

```
printf("\n p1 : %d p2: %d \n",p1,p2);
```

**Output :**

P1 = 6;

P2 = 5 ;

If "char \*str = "smartplay";int p1 = sizeof(str);" then p1 is 8. Pointer size 8 will be return.

Please check Q4 again.

Hi! All,

Below are the list of questions discussed on C-session (Apr 16 & 17).

| Q.No | Author | Question                                                                                                 | Output                                                                                                                                            | Discussion                                                                                                                                         |
|------|--------|----------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------|
|      |        | <pre>char *const ptr; char ch1,ch2; ptr = &amp;ch1; ptr = &amp;ch2;  What is value stored in ptr ?</pre> | <pre>Compilation Error: day1.c:6:1: error: assignment of read-only variable ‘ptr’ day1.c:7:1: error: assignment of read-only variable ‘ptr’</pre> | <pre>char *const ptr; ==&gt; Address is constant , value can be changed. Char const *ptr; ==&gt; Value is constant , address can be changed.</pre> |
| 2    |        | <pre>char arr[7] = “HELLO123”; printf("\n arr : %s",arr); printf("\n arr2 %c \n",*arr);</pre>            | <pre>arr : HELLO12 arr2 : H</pre>                                                                                                                 |                                                                                                                                                    |
| 3    |        | <p>Explain situation to use double pointer</p>                                                           | <pre>int *p; func(&amp;p);  void func(int **q){ *q = malloc(sizeof(int)); }</pre>                                                                 |                                                                                                                                                    |

|   |   |                                                                                                                                                                                  |                                                                    |                                                                                                                                                 |
|---|---|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------|
| 4 |   | What the value of p1 & p2;<br>char *str = "smartplay";<br>int p1 = sizeof(str);<br>Int p2 = strlen(str);<br><br>printf("\n p1 : %d p2: %d<br>\n",p1,p2);                         | P1 = 6;<br>P2 = 5 ;                                                | Sizeof operator returns the size of string including null character<br><br>Strlen function returns length of a string excluding null character. |
| 5 | a | char *str1 = "smartplay";<br>char *str2[] = "smartplay";<br><br>printf("\n str1 : %u str2: %u<br>\n",sizeof(str1),sizeof(str2));                                                 | Compilation error at<br>*str2[].<br><br>error: invalid initializer | str2 pointer should be memory allocated.<br><br>Char *str2 = (char *) malloc(sizeof(5));<br><br>Do dynamic allocation to avoid such errors.     |
| 6 | D | #define TRUE 0<br>#define FALSE -1<br>#define NULL 0<br><br>void main(){<br><br>if(TRUE) printf("TRUE");<br><br>else if(FALSE) printf("FALSE");<br><br>Else printf("NULL");<br>} | FALSE                                                              | -1 = FF Consider as TRUE                                                                                                                        |

|   |                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |                     |                                                                                                                                                                                                                               |
|---|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 7 | <pre>enum actor {     abc = 5,     xyz = -2,     gcd,     hcf };  void main(){      enum actor a =0;     switch(a) {         case abc:             printf("abc");             break;         case xyz:             printf("xyz");             break;         case gcd:             printf("gcd");             break;         case hcf:             printf("hcf");             break;         default:             printf("default");             break;     } }</pre> | hcf                 | xyz = -2,<br>gcd, // gets -1 (xyz + 1)<br>hcf // gets 0 ( gcd +1)                                                                                                                                                             |
| 8 | <pre>switch(*(1 + "AB" "CD" + 1)) {     case 'A':         printf("A");         break;     case 'B':         printf("B");         break;     case 'C':         printf("C");         break;     case 'D':         printf("D");         break;     default:         printf("default");         break; }</pre>                                                                                                                                                            | C                   | 1 +ABCD + 1<br>2 + ABCD<br><br>A = 0 , B = 1 C = 2 , D = 3<br><br>Expected value is 2 , So it reach case C:<br><br>If switch((1 + "AB" "CD" + 1))<br>then its an compilation error :<br>error: switch quantity not an integer |
| 9 | <pre>struct student {     int rno;     char name[10]; }st;  void main(){      st.rno = 10;     <u>st.name</u> = "hello"; }</pre>                                                                                                                                                                                                                                                                                                                                      | Compilation Error : | <u>st.name</u> = "hello"; is not allowed<br><br>strcpy( <u>st.name</u> ,"hello"); is allowed                                                                                                                                  |

|    |   |                                                                                             |                                                                                                                                              |                         |
|----|---|---------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------|-------------------------|
| 10 | G | <pre>int xx = NULL; int yy = 10; int zz = 0;  zz = xx + yy;  printf("Z : %d \n ",zz);</pre> | Z = 10;                                                                                                                                      | NULL Considered as a 0. |
| 11 |   | <pre>printf("char : %c ",59); printf("int : %d ",59); printf("string : %s ",59);</pre>      | Char - ; Int - 59 String - Segmentation fault                                                                                                |                         |
| 1  |   | <pre>int i = 4,5,6; Int I = (4,5,6);</pre>                                                  | First statement - Not allowed - error: expected identifier or '(' before numeric constant.<br><br>Second statement - I = 6 will be assigned. |                         |
| 13 |   | <pre>int I = -1; while(+(+i) != 0) { ++i; }</pre>                                           | I = 0;                                                                                                                                       |                         |

**U**

sorry for wrong answer in 1st question. please check it.

Q 1) if else statement is better than switch case if yes then why?

switch statement is better than if-else, because compiler is better in optimizing a switch-statement than an if-statement.

The compiler doesn't know if the order of evaluating the if-statements is important to you, and can't perform any optimizations there. You could be calling methods in the if-statements, influencing variables. With the switch-statement it knows that all clauses can be evaluated at the same time and can put them in whatever order is most efficient.

Q 1) if else statement is better than switch case if yes then why?

If else statement is better than switch, because compiler is better in optimizing a switch-statement than an if-statement.

The compiler doesn't know if the order of evaluating the if-statements is important to you, and can't perform any optimizations there. You could be calling methods in the if-statements, influencing variables. With the switch-statement it knows that all clauses can be evaluated at the same time and can put them in whatever order is most efficient.

Q 2)

```
main()
{
int i=5;
change();
int i=10;
```

```
printf("%d",i)
}
change()
{
?(need to write code here)
}
```

in above statement print the 5 with respect to the change function.

Ans. #define printf(a,b) print("5");

Description:

defined in above of main function is also It Prints 5.

Printf function is a Built in function Even it can be used as a User defined function.

Q 3)

```
#define printd(expr) printf(#expr " %d",expr)
main()
{
int a=10,b=2;
printd(a/b);
}
```

What is o/p?

Ans: a/b=5;  
#expr treats that variable names.

Q.1 write a function which takes 7 as input and returns 4 and vice versa ?

```
A.1 int func1(int p) // input p should be 7 or 4
{
int i;
i=(p^3);
return i;
}
```

Q.2 what is stack flow after the statement being executed ?

OR

How these four things gets into picture 1. Compiler 2. C Language 3. Glibc 4. Kernel after the below statement?

```
char *p=((char *)malloc (10));
```

A.2

1. C Language :- It specifies the header file <stdlib.h> ,which will check the prototype.

2. Compiler :- It takes the output of the preprocessor, and the source code, and generates assembler source

code it is kept in an object file.

3. Glibc :- The glibc package contains standard libraries. In order to save disk space and memory, as well as to make upgrading easier, common system code is kept in one place and shared between programs. This particular package contains the most important sets of shared libraries: the standard C library and the standard math library. Without these two libraries, a Linux system will not function

So object file generated now gets linked with the Glibc where in implementation of malloc is being defined.

There are other APIs like uclibc , bionic libc which can be linked with different OS (Linux, BCD, QNX) in .Config file while building kernel, depending on the requirement of the user.

3. Kernel :- Glibc calls the kernel via brk() system call.

1. Allocated 2D memory with malloc.

Ans: Pointers can be easily used to create a 2D array in C using malloc. The idea is to first create a one dimensional array of pointers, and then, for each array entry, create another one dimensional array. Here's a sample code:

```
double**theArray;
theArray=(double**)malloc(arraySizeX*sizeof(double*));
for(int i=0; i<arraySizeX;i++)
theArray[i]=(double*)malloc(arraySizeY*sizeof(double));
```

-----  
2.

```
int * fun()
{
static int i;
i=20;
return (&i);
}
```

Ans: its depend on OS implementation because static variable store in Data segment but it will try to return the value from stack so some times it works some time it does not work because it is dependent on how OS have property of memory distribution.

-----  
3.

```
int fun(int x)
{
int y;
y=x-1;
return((x&y)==0);
}
```

Ans: if value is 2 power n then it will return 1 else return 0 ;

sol: case 1: x=4 and then y=4-1=3

x=0100 and y =0011

So ((0100 & 0011)==0) it is true it will return 1.

Case2: x=6 and the y=5

x=0110 and y=0101  
So ( (0110 & 0101)==0) it is not true so it will return 0.  
conclusion: if value is 2 power n then it will return 1 else return 0 ;

---

Subject: RE: C Sessions: Day-4 Questions

C session (Tuesday,09.04.2013)

Que- How can we pass entire array (all element of array) to the function.

Ans- for passing an array, wrap it in a structure:

for example-

```
struct xyz
{
int array[100];
};

int somefunc(struct xyz large)
{
...
}
void anotherfunc(void)
{
struct xyz a;
printf("%d\n", somefunc(a));
}
#####
#####
```

Que- What is the O/P of following program?

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

```

main()
{
char arr[]="string";

printf("%c",*arr);

arr++;
printf("%c",*arr);

}

```

Ans- Error !!!  
error Line 9 : '++' needs l-value

A side effect of arr++ is an attempt to change the value of arr.we can't use statement like , 'arr++' because , arr is a constant pointer to the allocated memory

```
#####
#####
```

Q.1 predict the output for this code snippet with reason?

```

main()
{
int (*ptr)[5];

char *p;

int arr[5]={1,2,3,4,5};
ptr=arr;
p=arr;
printf("ptr=%u \n p=%u",ptr,p);
p++;
ptr++;
printf("ptr=%u \n p=%u",ptr,p);

printf("*p=%u\n *ptr=%u",*p,*ptr);

printf("**ptr=%u\n",**ptr);

}

```

A.1 ptr=2402524240 p=2402524240

ptr=2402524260 p=2402524241 ( ptr is pointer to array so increments by 20 bytes int size x array size , while p is pointer to char so increments by 1 byte)

\*p=0 ( p is char pointer so it takes first byte of 4 bytes assigned that is zero here

0000 0000 0000 0001 , value can be diff if it is big endian)  
\*ptr=2402524260 ( single dereference cant give u value as it is pointer to array,we have to double dereference it for getting value)

\*\*ptr= 0; ( it gives value to next array memory is not assigned so it can take any value)

#####

Que- X % Y (X modulus Y )

Given- y is a power of 2.

calculate X % Y without using % operator.

Solution-

x modulo y = (x & (y - 1))

#####

Que- int a= some bit stream;

int b = another bit stream;

only 1 bit difference between 'a' and 'b'. find out the position of mismatch bit.

Ans-

```
#include <iostream>
using namespace std;
int main()
{
int a=1123,b=1091; // declared a and b with integer and difference of their bits is only 1
int c,d;
int count=0;
c= a ^ b ; //
d= c-1;
if((c & d) ==0)
{
while(c !=0)
{
count++;
c=(c>>1);
}
}
cout<<count; // return position where bits are mismatched
}
```

#####

Que-

I have a bit stream and without using for loop, I want to set m bit to n bit.

(m >n)

eg:

Bitstream = 10 10 10 10 01 00

m= 10

n=4,

O/P- 00 01 11 11 00 00

Solution :

(-1 << n) & ((1<<m) - 1)

Note: given solution set all the bit within the range but turn off the bits which are beyond the range.  
to overcome this problem we can use this logic.

```
num | (((1<<(m- n+1)-1) << (n-1))
```

```
#####
```

Subject: C Sessions: Day-4 Questions

strtok

```
char *strtok(char *str, const char *delim);
```

The strtok() function parses a string into a sequence of tokens. On the first call to strtok() the string to be parsed should be specified in str. In each subsequent call that should parse the same string, str should be NULL. The delim argument specifies a set of bytes that delimit the tokens in the parsed string. The caller may specify different strings in delim in successive calls that parse the same string. Each call to strtok() returns a pointer to a null-terminated string containing the next token. This string does not include the delimiting byte. If no more tokens are found, strtok() returns NULL.

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

```
int main(){
char str[]="hello-this is rider";
char *pch;
char* a[10];
int i=0,j;
```

```

pch=strtok(str," ");
//we can add more special character

while(pch!=NULL){
printf("%s\n",pch);
a[i]=pch;
i++;
pch=strtok(NULL," ");
}

reverseWords(a,i);
printf("Reverse:\n");

for(j=0;j<i;j++)
printf("%s\n", a[j]);
return 0;
}

```

O/P:

```

rider
is
hello-this

```

```

void main()
{
int a[2][3][2] = {{{2,4},{7,8},{3,4}},{ {2,2},{2,3},{3,4}}};
printf("0x%u 0x%u 0x%u %d \n",a,*a,**a,***a);
printf("0x%u 0x%u 0x%u %d \n",a+1,*a+1,**a+1,***a+1);
}

```

O/P:

```

0x382857360 0x382857360 0x382857360 2
0x382857384 0x382857368 0x382857364 3

```

Explanation:

The base address of array is 0x382857360.

As we know element at location [1][1][1] can be accessed as “a[1][1][1]” or \*( \*( \*(a + 1) + 1) + 1).

T \*p; // p is a pointer to an object of type T

When a pointer p is pointing to an object of type T, the expression \*p is of type T.

For example a is of type array of 2 two dimensional arrays.

The type of the expression \*a is “array of arrays (i.e. two dimensional array)”.

Based on the above concept translating the expression \*( \*( \*(buffer + 1) + 1) + 1).

For a three-dimensional array, reference to an array has type "pointer to array of 3 ints, of which each element as an array of 2 ints".

Therefore, a+1 is pointing to the memory location of first element of the second 2D array, a.

So,

a - An array of 2 two dimensional arrays, i.e. its type is “array of 2 two dimensional arrays”.

a + 1 - displacement for 2nd element in the array of 5 two dimensional arrays.

i.e  $a+1$  is equivalent to  $a[1][0][0]$

Hence  $382857360 + 3 * (2 \text{ ints} * 4 \text{ bytes}) = 382857384$

Similarly, for  $*a+1$

$*(a + 0) + 1$  - displacement to access 1th element in the array of 3 one dimensional arrays.

Therefore  $*a+1$  is equivalent to  $a[0][1][0]$

Similarly, for  $**a+1$

$*(*(a + 0) + 0) + 1$  - displacement to get element at 1th position in the single dimension array of integers.

Therefore  $**a+1$  is equivalent to  $a[0][0][1]$

Similarly, for  $***a+1$  is equivalent to value at  $a[0][0][1]$

$*(*(*(a + 0) + 0) + 0)$  - accessing the element at 0th position (the overall expression type is int now).

### String Literals

```
char *string = "English"; /* string literal */
char array[] = "English"; /* Char array not a string literal */
array[0] = 'V' ; /*Legal, no error*/
string[0] = 'V'; /* Undefined */
```

ANSI C standard defines String literals as " A string has type ``array of characters" and storage class static and is initialized with the given characters. Whether identical string literals are distinct is implementation defined, and the behavior of a program that attempts to alter a string literal is undefined." Trying to modify string literal will give segmentation fault on Linux however it may work on other OS or on processors which do not have memory protection.

In gcc/Linux string literal [char \*ptr] is stored read-only area hence you get a segmentation fault. But if non gcc compiler stores string literal in Read-Write area OR if the processor does not have MMU and OS does not offer memory protection this program will work without any error. This is because C language standard does not clearly define what should be result of code trying to modify string literals; C standard had left it to compilers/OS/Processor to use in best way they can.

Note that

### Structure alignment

1. Memory is always byte addressable.
2. For speed optimization compiler by default uses alignment which is same as data bus size. For ex. In case of machine with 32-bit data bus alignment is done in multiples of 4-bytes.
3. In case of data types smaller than size of data bus padding bytes are added so that each variable is aligned.
4. Problem with padding: In real communication and in storage padding is never used as it is unnecessary wastage of time & space. But if the padded structures used to reference protocol header or in media files data it could lead to incorrect results because of difference in alignment of actual data verses padded structures.
5. To avoid such issues compilers provide #pragma pack [see compiler document for specific syntax]. With this pragma programmer can instruct compiler not to add any padding and all the structures must be packed even though it mean compromise of efficiency.

### Precedence & Associativity

```
#include<stdio.h>
main()
{
int a =10, b=20, c=0;
c=a+++b; /*No spaces in between*/
printf("a=%d, b=%d, c=%d\n",a,b,c);

return 0;
}
```

ANS:a=11, b=20 c=30

1. This is because ++ has higher precedence than +
  2. ++ has "Right to Left" associativity hence it gets associated with a and becomes a++ i.e. post increment.
- 
- 

### String Literals

```
char *string = "English"; /* string literal */
char array[] = "English"; /* Char array not a string literal */
array[0] = 'V' ; /*Legal, no error*/
string[0] = 'V'; /* Undefined */
```

ANSI C standard defines String literals as " A string has type ``array of characters" and storage class static and is initialized with the given characters. Whether identical string literals are distinct is implementation defined, and the behavior of a program that attempts to alter a string literal is undefined." Trying to modify string literal will give segmentation fault on Linux however it may work on other OS or on processors which do not have memory protection.

In gcc/Linux string literal [char \*ptr] is stored read-only area hence you get a segmentation fault. But if non gcc compiler stores string literal in Read-Write area OR if the processor does not have MMU and OS does not offer memory protection this program will work without any error. This is because C language standard does not clearly define what should be result of code trying to modify string literals; C standard had left it to compilers/OS/Processor to use in best way they can.

Note that

### Structure alignment

1. Memory is always byte addressable.
2. For speed optimization compiler by default uses alignment which is same as data bus size. For ex. In case of machine with 32-bit data bus alignment is done in multiples of 4-bytes.
3. In case of data types smaller than size of data bus padding bytes are added so that each variable is aligned.
4. Problem with padding: In real communication and in storage padding is never used as it is unnecessary wastage of time & space. But if the padded structures used to reference protocol header or in media files data it could lead to incorrect results because of difference in alignment of actual data verses padded structures.
5. To avoid such issues compilers provide #pragma pack [see compiler document for specific syntax]. With this pragma programmer can instruct compiler not to add any padding and all the structures must be packed even though it means compromise of efficiency.

Precedence & Associativity

```
#include<stdio.h>
main()
{
int a =10, b=20, c=0;
c=a+++b; /*No spaces in between*/
printf("a=%d, b=%d, c=%d\n",a,b,c);

return 0;
}
```

ANS:a=11, b=20 c=30

1. This is because ++ has higher precedence than +
2. ++ has "Right to Left" associativity hence it gets associated with a and becomes a++ i.e. post increment.

//

I2C protocol in details with timing diagrams

SPI protocol in details with timing diagrams

What is interrupt latency? How to reduce it?

What is difference between exceptions and interrupts?

Write C function to extract bits from position m to n where m > n and toggle those bits.

Write optimized function to convert little endian to big endian.

Write function to find if an integer is power of 2 or not

Write test cases for add function which adds three input values.

Different types of testing done during project work

Explain details of projects worked on so far. More questions on project.

Write and explain architecture of any latest project worked

Different modes in ARM processor? When is each mode entered?

What is system call? How are parameters passed in system call?

How are nested interrupts handled?

Explain complete boot process till linux is booted from power on reset.

Many questions on internals of boot process.

What are different boot devices available?

What are inputs for testing?

What test coverage tools have been used so far in projects worked?

1.What is volatile ?

2.What are Message queue & shared memory?

Which one is better explain with some example.

3.How to protect shared memory ?

4.Insert a node at a given position in a Linked list.

5.Endianness : How int i=10 stored in Little endian format

Which architecture uses Big endian.

int i,\*ptr;

ptr=&i;(Assuming &i is 1000)

ptr = ptr + 10;

What will be the value present in ptr.

ans : 1040

How to round off a number to a multiple of 1024 x 1024 (1Mb)(Using shifting operations)

6) int i;

Should be used in different files like a.c b.c c.c. How to achieve it.

7) Use of virtual memory

8) Which module of virtual memory does memory protection.

9) Delete a node from a double linked list pseudo code.

1. what is "little endian" and "big endian" and how to know this?

2. Is it possible to specify width of the data in strings? how? and what happens when that exceeded or reduced

3. what are the "unary operators?"

4. Is it possible to assaign a value to a variable before declaring?

5. can you write an inline code for finding largest of three numbers using ternary operator?

6. what is inline function in c?

7. write a program for '/' and '%' operators without using them.

8. Difference between return and goto functions

9. write a program to print pascal triangle

10. where main function return will go

11. where function declarations are present

12. who calls the main() function

13. Difference between recursion and iteration

14. Is it possible to multiply or division of two pointers?what are the arithmetic operations supported in pointers.

What is mutex ? why mutexes?

- 14.Why spinlocks are so important whats special with it?
- 15.Explain Priority Inversion with an example?
- 16.How do you handle events in Linux? Do You know about Signals and its handling?
- 17.What are ARM modes?
- 18.Why banked registers?
- 19.When Can u expect Stack Corruption?
- 20.How Can u decide the stack size?
- 21.Static and Global Variables ? Why ?
- 22.What is heap?
- 25.Write the code for Interrupt handling?
- 26.What are Scheduling Policies?
- 27.What are Cache Policies?Explain ?
- 28.What is the Size of Cache Line?

Write a C program to count number of 1's in a number?

What is semaphore , mutex and spinlock?

In ISR which synchronization mechanism is used and why?

Advantage of mutex over semaphore?

What is deadlock?

Can we detect deadlock?

What is priority inversion and how to avoid it?

In which cases deadlock situation occurs?

Write a program to find out sizeof a variable with out using sizeof operator?

What is malloc() and calloc()

Differences between malloc and calloc?

What is paging?

Form which area is dynamically allocated using malloc and whether that memory is virtual contiguous or virtual non contiguous or physical contiguous or physical non contiguous?

In kernel which function is used to allocate memory dynamically?

What is the use of wireshark?

What is USB?

What are the layers of USB in host side and client side?

32. fork()

fork()

fork

What is ISR?

7. Tell me what will happen when you are serving one interrupt and another interrupt comes.

8. Write a program for const variable? can you change the value of the const variable in program?  
write the program?

9.what is little endian and Big endian and write a program to in which 32 bit no changed into big endian?

10.memory leaks?

12. How to avoid memory leak?

13. What is malloc, write prototype, what it returns?

14. What will happen when you are accessing NULL pointer.

15. What is linked list?

16. Give a scenario in which loop detects in Linked list?
  17. How to avoid and detect loop in linked list?
  18. Linked list traversal?
  19. What is bottom halves?
  20. Give an example where we use Mutex and Spin locks?
  21. Tell the difference between Mutex and spin locks?
- Implement sizeof macro.it has to work if you give any variable.(ptr,int,var).
- 

\*\*\*\*\*

## Intel F2F (video conference) Interview Questions

### For Validation requirement

1. Can u brief about your experience
2. How many projects you have worked on Linux
3. Explain about static with an example(using board)
4. Can we declare a global variable after main, if not why
5. Can we extend the scope of a static variable to other file
6. Tell me any real time example where you have used static for functions and variables
7. Can we return address from a function ,whether it will give any error
8. Explain about **malloc()**
9. Write a program for function pointer, what's its advantage
10. Can we use the combination of **static** with **extern**
11. Explain about your Linux project
12. How you are interacting with kernel space from user space
13. What are the system calls you have used so far
14. Can you explain any one system call with an example
15. What is **ioctl** in Linux driver , have you used it
16. How your getting data from battery monitoring IC
17. Explain about DMA
18. How many channels you have used in DMA
19. How to transfer 36 bytes of data using a fixed burst size and burst length
20. Explain any one test case in DMA
21. How you are configuring DMA
22. Explain any problem that you have faced during DMA data transfer

23. Your DMA is related to your UART or what exactly
  24. What is the difference between baud rate and bit rate
  25. How you are configuring your UART for a particular frequency
  26. Have you done Python scripting
  27. Do you know CAN protocol
  28. What have you done in USB 3.0 till now
  29. Tell me some differences that you found from USB 3.0 to previous versions
  30. What is polling
  31. How can you avoid polling in USB 3.0
  32. What is backward compatibility in USB 3.0
  33. How host will find whether the device is high speed device in USB 3.0
  34. What is speed negotiation
  35. How hub will react with high speed device
  36. What is K chirp and JK chirp that you have told
  37. Tell me difference between SD, eMMC and SDIO
  38. Can you tell me example of SDIO
  39. Can list some eMMC commands which differ from SD
  40. Explain initialization sequence of SD
  41. What is Sanitize in eMMC
  42. If I give a command which is not there in SD, what happens
- .....
- .....

Dear All,

C Questions discussed on 22 April 2013 are listed below. I don't know upto what extend these solutions are correct. Please let me know if they are wrong

//Q1 Traverse binary tree breadth-wise

//Ans:

```
void print(BinaryTree *root)
```

```
{
```

```
 if (!root)
```

```
 return;

queue<BinaryTree*> currentLevel, nextLevel;
currentLevel.push(root);
while (!currentLevel.empty())
{
 BinaryTree *currNode = currentLevel.front();
 currentLevel.pop();
 if (currNode)
 {
 printf(currNode->data) ;
 nextLevel.push(currNode->left);
 nextLevel.push(currNode->right);
 }
 if (currentLevel.empty())
 {
 printf("end");
 swap(currentLevel, nextLevel);
 }
}
```

```
//find out least common ancestor in tree
void find(root, node1 , node2)
{
 if(root==null | | node1 ==null | | node2==null)
 return null;
 if(node1== root)
```

```
 return node1;
if(node2==root)
 return node2;
if(node1 < root && node2 < root)
{
 return find(root->left,node1,node2);
}
elseif(node1 > root && node2 > root)
{
 return find(root->right,node1,node2);
}
return root;
}
```

```
//Dear All,
//These are the C Questions discussed on 19-04-2013, listed below.
//Q1.) Reverse a string using bitwise operator?
//Ans: using xor operator.

Int main()
{
 Char x[] = "Manohar";
 Char *string = x;
 int len = strlen(string);
 for (int i=0; i<len/2; i++)
 {
```

```

 x[len-i-1] ^= x[i];
 x[i] ^= x[len-i-1];
 x[len-i-1] ^= x[i];
}

}

```

Q2.) printf("%d", -1 <<4); what is the output ?

Ans: 1 is represented in binary form is 0000 0000 0000 0001

-1 means 1's compliment                    1111 1111 1111 1110

Output is FFF0

Q3.) Use of bit fields in C?

Ans: Both C and C++ allow integer members to be stored into memory spaces smaller than the compiler would ordinarily allow. Width in bit fields explicitly declared and A bit field declaration may not use either of the type qualifiers, const or volatile.

The following restrictions apply to bit fields. You cannot:

- &#8226; Define an array of bit fields
- &#8226; Take the address of a bit field
- &#8226; Have a pointer to a bit field
- &#8226; Have a reference to a bit field

The following example demonstrates padding. The structure kitchen contains eight members totaling 16 bytes.

```

struct on_off {
 unsigned light : 1;
 unsigned toaster : 1;
 int count; /* 4 bytes */
 unsigned ac : 4;
 unsigned : 4;
 unsigned clock : 1;
 unsigned : 0;
 unsigned flag : 1;
} kitchen ;

```

Member Name

Storage Occupied

light

1 bit

toaster

1 bit

(padding -- 30 bits)

To the next int boundary

count

The size of an int (4 bytes)

ac

4 bits

(unnamed field)

4 bits

clock

1 bit

(padding -- 23 bits)

To the next int boundary (unnamed field)

flag

1 bit

(padding -- 31 bits)

To the next int boundary

Q4.) what is the output of given code ?

Int x = 5;

Printf("%d %d %d",x, x<<2, x>>2);

Ans: 5 20 1

Q5.) char \*ptr=NULL;

Char arr[10]={"abcd"};

Strcpy(ptr,arr);

```
Printf("%s",ptr);
```

What is the output?

Ans: segmentation fault. Since pointer is not pointing to any valid memory.

Q6.) Int x[]={1,2,3,4,5};

```
Int *ptr, **ptr2;
```

```
Ptr = x;
```

```
Ptr2 = &ptr ;
```

How do you update x[2]= 10 using ptr2 ?

Ans: ptr2 + 2 and \*\*ptr = 10;

Or

```
**(ptr2 + 2) = 10;
```

Q7.) Convert the expression ((A + B) \* C - (D - E) ^ (F + G)) to postfix notation.

Ans: AB + C \* DE - - FG + ^

Q8.) Predict the output from the given code?

```
Struct abc{
```

```
int a;
```

```
float b;
```

```
}d;
```

```
main(){
```

```
 d.a = 1;
```

```
 printf("%f" , d.b);
```

```
}
```

Ans: 0.0

Q9.) Predict the which one is not valid statement?

1.++d+++e++;

2.f\*=g+=h=5;

3.L-- >> M << -- N;

4.int a(int a), b = 0, c =a((c=b,++b));

5.i- >j< -k

Ans: 5

Thank you!

Best Regards,

Manohar

Q4:

```
int p1 = sizeof("HELLO");
Int p2 = strlen("HELLO");

printf("\n p1 : %d p2: %d \n",p1,p2);
```

Output :

P1 = 6;

P2 = 5 ;

If "char \*str = "smartplay";int p1 = sizeof(str);" then p1 is 8. Pointer size 8 will be return.

Please check Q4 again.

Hi! All,

Below are the list of questions discussed on C-session (Apr 16 & 17).

Q.No

Author

Question

Output

Discussion

```
char *const ptr;
char ch1,ch2;
ptr = &ch1;
ptr = &ch2;
```

What is value stored in ptr ?

Compilation Error:

```
day1.c:6:1: error: assignment of read-only variable 'ptr'
day1.c:7:1: error: assignment of read-only variable 'ptr'
char *const ptr; ==> Address is constant , value can be changed.
Char const *ptr; ==> Value is constant , address can be changed.
```

2

```
char arr[7] = "HELLO123";
printf("\n arr : %s",arr);
printf("\n arr2 %c \n",*arr);
arr : HELLO12
arr2 : H
```

3

Explain situation to use double pointer

```
int *p;
func(&p);

void func(int **q){
*q = malloc(sizeof(int));
}
```

4

What the value of p1 & p2;

```
char *str = "smartplay";
```

```
int p1 = sizeof(str);
```

```
Int p2 = strlen(str);
```

```
printf("\n p1 : %d p2: %d \n",p1,p2);
```

```
P1 = 6;
```

```
P2 = 5 ;
```

Sizeof operator returns the size of string including null character

Sstrlen function returns length of a string excluding null character.

5

```
a
```

```
char *str1 = "smartplay";
```

```
char *str2[] = "smartplay";
```

```
printf("\n str1 : %u str2: %u \n",sizeof(str1),sizeof(str2));
```

Compilation error at \*str2[].

error: invalid initializer

str2 pointer should be memory allocated.

```
Char *str2 = (char *) malloc(sizeof(5));
```

Do dynamic allocation to avoid such errors.

6

D

```
#define TRUE 0
#define FALSE -1
#define NULL 0
```

```
void main(){
```

```
 if(TRUE) printf("TRUE");

 else if(FALSE) printf("FALSE");

 Else printf("NULL");
```

```
}
```

FALSE

-1 = FF Consider as TRUE

7

```
enum actor {
```

```
 abc = 5,
```

```
 xyz = -2,
```

```
 gcd,
```

```
 hcf
```

```
};
```

```
void main(){
```

```
 enum actor a =0;
```

```
 switch(a) {
```

```
 case abc:
```

```
printf("abc");
break;

case xyz:
printf("xyz");
break;

case gcd:
printf("gcd");
break;

case hcf:
printf("hcf");
break;

default:
printf("default");
break;
}

hcf
xyz = -2,
gcd, // gets -1 (xyz + 1)
hcf // gets 0 (gcd +1)
8
```

```
switch(*(1 + "AB" "CD" + 1)) {

case 'A':
printf("A");
break;

case 'B':
printf("B");
break;

case 'C':
```

```
printf("C");
break;
case 'D':
printf("D");
break;
default:
printf("default");
break;
}
C
1 +ABCD + 1
2 + ABCD
```

A = 0 , B = 1 C = 2 , D = 3

Expected value is 2 , So it reach case C:

If switch((1 + "AB" "CD" + 1)) then its an compilation error : error: switch quantity not an integer

9

```
struct student {
int rno;
char name[10];
}st;
```

```
void main(){
```

```
 st.rno = 10;
```

```
st.name = "hello";
```

Compilation Error :

error: incompatible types when assigning to type ‘char[10]’ from type ‘char \*’

st.name = "hello"; is not allowed

strcpy(st.name,"hello"); is allowed

10

G

```
int xx = NULL;
```

```
int yy = 10;
```

```
int zz = 0;
```

```
zz = xx + yy;
```

```
printf("Z : %d \n ",zz);
```

```
Z = 10;
```

NULL Considered as a 0.

11

```
printf("char : %c ",59);
```

```
printf("int : %d ",59);
```

```
printf("string : %s ",59);
```

Char - ;

Int – 59

String – Segmentation fault

1

```
int i = 4,5,6;
```

```
Int l = (4,5,6);
```

First statement – Not allowed - error: expected identifier or '(' before numeric constant.

Second statement – l = 6 will be assigned.

13

```
int l = -1;
```

```
while(+(+i) != 0) {
```

```
 ++i;
```

```
}
```

```
l = 0;
```

```
U
```

sorry for wrong answer in 1st question. please check it.

Q 1) if else statement is better than switch case if yes then why?

switch statement is better than if-else, because compiler is better in optimizing a switch-statement than an if-statement.

The compiler doesn't know if the order of evaluating the if-statements is important to you, and can't perform any optimizations there. You could be calling methods in the if-statements, influencing variables. With the switch-statement it knows that all clauses can be evaluated at the same time and can put them in whatever order is most efficient.

Q 1) if else statement is better than switch case if yes then why?

If else statement is better than switch, because compiler is better in optimizing a switch-statement than an if-statement.

The compiler doesn't know if the order of evaluating the if-statements is important to you, and can't perform any optimizations there. You could be calling methods in the if-statements, influencing variables. With the switch-statement it knows that all clauses can be evaluated at the same time and can put them in whatever order is most efficient.

Q 2)

```
main()
{
 int i=5;
 change();
 int i=10;
 printf("%d",i)
}
change()
{
 ?(need to write code here)
}
```

in above statement print the 5 with respect to the change function.

Ans. #define printf(a,b) print("5");

Description:

defined in above of main function is also It Prints 5.

Printf function is a Built in function Even it can be used as a User defined function.

Q 3)

```
#define printd(expr) printf(#expr "%d",expr)

main()
{
int a=10,b=2;
printd(a/b);
}
```

What is o/p?

Ans: a/b=5;

#expr treats that variable names.

Q.1 write a function which takes 7 as input and returns 4 and vice versa ?

A.1 int func1( int p ) // input p should be 7 or 4

```
{
int i;
i=(p^3);
return i;
}
```

Q.2 what is stack flow after the statement being executed ?

OR

How these four things gets into picture 1. Compiler 2. C Language 3. Glibc 4. Kernel after the below statement?

```
char *p=((char *)malloc (10));
```

A.2

1. C Language :- It specifies the header file <stdlib.h>, which will check the prototype.
2. Compiler :- It takes the output of the preprocessor, and the source code, and generates assembler source code it is kept in an object file.
3. Glibc :- The glibc package contains standard libraries. In order to save disk space and memory, as well as to make upgrading easier, common system code is kept in one place and shared between programs.

This particular package contains the most important sets of shared libraries: the standard C library and the standard math library. Without these two libraries, a Linux system will not function

So object file generated now gets linked with the Glibc where in implementation of malloc is being defined.

There are other APIs like uclibc, bionic libc which can be linked with different OS (Linux, BCD, QNX) in .Config file while building kernel, depending on the requirement of the user.

3. Kernel :- Glibc calls the kernel via brk() system call.

1. Allocated 2D memory with malloc.

Ans: Pointers can be easily used to create a 2D array in C using malloc. The idea is to first create a one dimensional array of pointers, and then, for each array entry, create another one dimensional array. Here's a sample code:

```
double**theArray;
theArray=(double**)malloc(arraySizeX*sizeof(double*));
for(int i=0; i<arraySizeX;i++)
theArray[i]=(double*)malloc(arraySizeY*sizeof(double));
```

---

2.

```
int * fun()
{
static int i;
i=20;
return (&i);
}
```

Ans: its depend on OS implementation because static variable store in Data segment but it will try to return the value from stack so some times it works some time it does not work because it is dependent on how OS have property of memory distribution.

---

3.

```
int fun(int x)
{
int y;
y=x-1;
return((x&y)==0);
}
```

Ans: if value is 2 power n then it will return 1 else return 0 ;

sol: case 1: x=4 and then y=4-1=3

x=0100 and y =0011

So  $((0100 \& 0011) == 0)$  it is true it will return 1.

Case2: x=6 and the y=5

x=0110 and y=0101

So  $( (0110 \& 0101) == 0)$  it is not true so it will return 0.

conclusion: if value is 2 power n then it will return 1 else return 0 ;

---

Subject: RE: C Sessions: Day-4 Questions

C session (Tuesday,09.04.2013)

Que- How can we pass entire array (all element of array) to the function.

Ans- for passing an array, wrap it in a structure:

for example-

```
struct xyz
{
 int array[100];
};

int somefunc(struct xyz large)
{
 ...
}

void anotherfunc(void)
{
 struct xyz a;
 printf("%d\n", somefunc(a));
}

#####
```

Que- What is the O/P of following program?

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

```
main()
{
 char arr[]="string";
```

```
 printf("%c",*arr);
```

```
 arr++;
}
```

```
 printf("%c", *arr);
```

```
}
```

Ans- Error !!!

error Line 9 : '++' needs l-value

A side effect of arr++ is an attempt to change the value of arr.we can't use statement like , 'arr++' because , arr is a constant pointer to the allocated memory

```
#####
#####
```

Q.1 predict the output for this code snippet with reason?

```
main()
```

```
{
int (*ptr)[5];

char *p;

int arr[5]={1,2,3,4,5};
ptr=arr;
p=arr;
printf("ptr=%u \n p=%u",ptr,p);
p++;
ptr++;
printf("ptr=%u \n p=%u",ptr,p);

printf("*p=%u\n *ptr=%u",*p,*ptr);

printf("**ptr=%u\n", **ptr);

}
A.1 ptr=2402524240 p=2402524240
```

ptr=2402524260 p=2402524241 ( ptr is pointer to array so increments by 20 bytes int size x array size , while p is pointer to char so increments by 1 byte)

\*p=0 ( p is char pointer so it takes first byte of 4 bytes assigned that is zero here

0000 0000 0000 0001 , value can be diff if it is big endian)

\*ptr=2402524260 ( single dereference cant give u value as it is pointer to array,we have to double dereference it for getting value)

\*\*ptr= 0; ( it gives value to next array memory is not assigned so it can take any value)

#####

Que- X % Y (X modulus Y )

Given- y is a power of 2.

calculate X % Y without using % operator.

Solution-

x modulo y = (x & (y - 1))

#####

Que- int a= some bit stream;

int b = another bit stream;

only 1 bit difference between 'a' and 'b'. find out the position of mismatch bit.

Ans-

```
#include <iostream>

using namespace std;

int main()

{

int a=1123,b=1091; // declared a and b with integer and diffrence of their bits is only 1

int c,d;

int count=0;

c= a ^ b ; //

d= c-1;

if((c & d) ==0)

{

while(c !=0)

{

count++;

c=(c>>1);

}

}

cout<<count; // return position where bits are mismatched

}
```

```
#####
#####
```

Que-

I have a bit stream and without using for loop, I want to set m bit to n bit.

(m >n)

eg:

Bitstream = 10 10 10 10 01 00

m= 10

n=4,

O/P- 00 01 11 11 00 00

Solution :

$(-1 << n) \& ((1 << m) - 1)$

Note: given solution set all the bit within the range but turn off the bits which are beyond the range.

to overcome this problem we can use this logic.

$num | ((1 << (m - n + 1)) - 1) << (n - 1)$

#####

Subject: C Sessions: Day-4 Questions

strtok

```
char *strtok(char *str, const char *delim);
```

The strtok() function parses a string into a sequence of tokens. On the first call to strtok() the string to be parsed should be specified in str. In each subsequent call that should parse the same string, str should be NULL. The delim argument specifies a set of bytes that delimit the tokens in the parsed string. The caller may specify different strings in delim in successive calls that parse the same string. Each call to strtok() returns a pointer to a null-terminated string containing the next token. This string does not include the delimiting byte. If no more tokens are found, strtok() returns NULL.

```
#include<string.h>

int main(){
 char str[]="hello-this is rider";
 char *pch;
 char* a[10];
 int i=0,j;
 pch=strtok(str," "); //we can add more special character

 while(pch!=NULL){
 printf("%s\n",pch);
 a[i]=pch;
 i++;
 }
}
```

```
pch=strtok(NULL," ");
}
```

```
reverseWords(a,i);
```

```
printf("Reverse:\n");
```

```
for(j=0;j<i;j++)

```

```
printf("%s\n", a[j]);

```

```
return 0;
}
```

O/P:

rider

is

hello-this

---

```
void main()
```

```
{
```

```
int a[2][3][2] = {{{2,4},{7,8},{3,4}},{ {2,2},{2,3},{3,4}}};
```

```
printf("0x%u 0x%u 0x%u %d \n",a,*a,**a,***a);
```

```
printf("0x%u 0x%u 0x%u %d \n",a+1,*a+1,**a+1,***a+1);
```

```
}
```

O/P:

0x382857360 0x382857360 0x382857360 2

0x382857384 0x382857368 0x382857364 3

Explanation:

The base address of array is 0x382857360.

As we know element at location [1][1][1] can be accessed as “a[1][1][1]” or  $\*(\*(\*(a + 1) + 1) + 1)$ .

T \*p; // p is a pointer to an object of type T

When a pointer p is pointing to an object of type T, the expression \*p is of type T.

For example a is of type array of 2 two dimensional arrays.

The type of the expression \*a is “array of arrays (i.e. two dimensional array)”.

Based on the above concept translating the expression  $\*(\*(\*(buffer + 1) + 1) + 1)$ .

For a three-dimensional array, reference to an array has type "pointer to array of 3 ints, of which each element as an array of 2 ints".

Therefore, a+1 is pointing to the memory location of first element of the second 2D array, a.

So,

a – An array of 2 two dimensional arrays, i.e. its type is “array of 2 two dimensional arrays”.

a + 1 – displacement for 2nd element in the array of 5 two dimensional arrays.

i.e a+1 is equivalent to a[1][0][0]

Hence  $382857360 + 3*(2 \text{ ints} * 4 \text{ bytes}) = 382857384$

Similarly, for \*a+1

$\*(a + 0) + 1$  – displacement to access 1th element in the array of 3 one dimensional arrays.

Therefore \*a+1 is equivalent to a[0][1][0]

Similarly, for  $**a+1$

$*( *(a + 0) + 0) + 1$  – displacement to get element at 1th position in the single dimension array of integers.

Therefore  $**a+1$  is equivalent to  $a[0][0][1]$

Similarly, for  $***a+1$  is equivalent to value at  $a[0][0][1]$

$*( *( *(a+ 0) + 0) + 0)$  - accessing the element at 0th position (the overall expression type is int now).

### String Literals

```
char *string = "English"; /* string literal */
char array[] = "English"; /* Char array not a string literal */
array[0] = 'V' ; /*Legal, no error*/
string[0] = 'V'; /* Undefined */
```

ANSI C standard defines String literals as " A string has type ``array of characters" and storage class static and is initialized with the given characters. Whether identical string literals are distinct is implementation defined, and the behavior of a program that attempts to alter a string literal is undefined." Trying to modify string literal will give segmentation fault on Linux however it may work on other OS or on processors which do not have memory protection.

In gcc/Linux string literal [char \*ptr] is stored read-only area hence you get a segmentation fault. But if non gcc compiler stores string literal in Read-Write area OR if the processor does not have MMU and OS does not offer memory protection this program will work without any error. This is because C language standard does not clearly define what should be result of code trying to modify string literals; C standard had left it to compilers/OS/Processor to use in best way they can.

Note that

### Structure alignment

1. Memory is always byte addressable.
2. For speed optimization compiler by default uses alignment which is same as data bus size. For ex. In case of machine with 32-bit data bus alignment is done in multiples of 4-bytes.
3. In case of data types smaller than size of data bus padding bytes are added so that each variable is aligned.
4. Problem with padding: In real communication and in storage padding is never used as it is unnecessary wastage of time & space. But if the padded structures used to reference protocol header or in media files data it could lead to incorrect results because of difference in alignment of actual data verses padded structures.
5. To avoid such issues compilers provide #pragma pack [see compiler document for specific syntax]. With this pragma programmer can instruct compiler not to add any padding and all the structures must be packed even though it mean compromise of efficiency.

### Precedence & Associativity

```
#include<stdio.h>

main()
{
 int a =10, b=20, c=0;
 c=a+++b; /*No spaces in between*/
 printf("a=%d, b=%d, c=%d\n",a,b,c);

 return 0;
}
```

ANS:a=11, b=20 c=30

1. This is because ++ has higher precedence than +
2. ++ has “Right to Left” associativity hence it gets associated with a and becomes a++ i.e. post increment.

//

I2C protocol in details with timing diagrams

SPI protocol in details with timing diagrams

What is interrupt latency? How to reduce it?

What is difference between exceptions and interrupts?

Write C function to extract bits from position m to n where m > n and  
toggle those bits.

Write optimized function to convert little endian to big endian.

Write function to find if an integer is power of 2 or not

Write test cases for add function which adds three input values.

Different types of testing done during project work

Explain details of projects worked on so far. More questions on project.

Write and explain architecture of any latest project worked

Different modes in ARM processor? When is each mode entered?

What is system call? How are parameters passed in system call?

How are nested interrupts handled?

Explain complete boot process till linux is booted from power on reset.

Many questions on internals of boot process.

What are different boot devices available?

What are inputs for testing?

What test coverage tools have been used so far in projects worked?

1.What is volatile ?

2.What are Message queue & shared memory?

Which one is better explain with some example.

3.How to protect shared memory ?

4.Insert a node at a given position in a Linked list.

5.Endianness : How int i=10 stored in Little endian format

Which architecture uses Big endian.

```
int i,*prt;
ptr=&i;(Assuming &i is 1000)
ptr = prt + 10;
```

What will be the value present in ptr.

ans : 1040

How to round off a number to a multiple of  $1024 \times 1024$  (1Mb)(Using shifting operations)

6) int i;

Should be used in different files like a.c b.c c.c. How to achieve it.

7) Use of virtual memory

8) Which module of virtual memory does memory protection.

9) Delete a node from a double linked list pseudo code.

1. what is "little endian" and "big endian" and how to know this?

2. Is it possible to specify width of the data in strings? how? and what happens when that exceeded or reduced

3. what are the "unary operators?"

4. Is it possible to assign a value to a variable before declaring?

5. can you write an inline code for finding largest of three numbers using ternary operator?

6. what is inline function in c?

7. write a program for '/' and '%' operators without using them.

8. Difference between return and goto functions

9. write a program to print pascal triangle

10. where main function return will go

11. where function declarations are present

12. who calls the main() function

13. Difference between recursion and iteration

14. Is it possible to multiply or division of two pointers?what are the arithmetic operations supported in pointers.

What is mutex ? why mutexes?

- 14.Why spinlocks are so important whats special with it?
- 15.Explain Priority Inversion with an example?
- 16.How do you handle events in Linux? Do You know about Signals and its handling?
- 17.What are ARM modes?
- 18.Why banked registers?
- 19.When Can u expect Stack Corruption?
- 20.How Can u decide the stack size?
- 21.Static and Global Variables ? Why ?
- 22.What is heap?
- 25.Write the code for Interrupt handling?
- 26.What are Scheduling Policies?
- 27.What are Cache Policies?Explain ?
- 28.What is the Size of Cache Line?

Write a C program to count number of 1's in a number?

What is semaphore , mutex and spinlock?

In ISR which synchronization mechanism is used and why?

Advantage of mutex over semaphore?

What is deadlock?

Can we detect deadlock?

What is priority inversion and how to avoid it?

In which cases deadlock situation occurs?

Write a program to find out sizeof a variable with out using sizeof operator?

What is malloc() and calloc()?

Differences between malloc and calloc?

What is paging?

Form which area is dynamically allocated using malloc and whether that memory is virtual contiguous or virtual non contiguous or physical contiguous or physical non contiguous?

In kernel which function is used to allocate memory dynamically?

What is the use of wireshark?

What is USB?

What are the layers of USB in host side and client side?

32. fork()

fork()

fork

What is ISR?

7. Tell me what will happen when you are serving one interrupt and another interrupt comes.

8. Write a program for const variable? can you change the value of the const variable in program? write the program?

9.what is little endian and Big endian and write a program to in which 32 bit no changed into big endian?

10.memory leaks?

12. How to avoid memory leak?

13. What is malloc, write prototype, what it returns?

14. What will happen when you are accessing NULL pointer.

15. What is linked list?

16. Give a senario in which loop detects in Linked list?

17.How to avoid and detect loop in linked list?

18.Linked list traversal?

19. What is bottom halfs?

20. Give an example where we use Mutex and Spin locks?

21. Tell the difference between Mutex and spin locks?

Implement sizeof macro.it has to work if you give any variable.(ptr,int,var).

---

### **1.What is job of preprocessor, compiler, assembler and linker ?**

### **2.What is interrupt latency?**

3. How to decide whether given processor is using little endian format or big endian format ?

**4.What is Top half & bottom half of a kernel?**

**5.How many types of IPC mechanism you know?**

**6.What is difference between binary semaphore and mutex?**

**7.What is interrupt?**

**8.What is cache memory?**

9.What are volatile variables?

10.What does static variable mean in C?

11.Explain the use of bit field.

12.What are Pipes? Explain use of pipes.

13.How does the linux file system work?

14.What is a semaphore?

15.What is spin lock?

16.Compare Thread and process

**17.Explain about ioctl**

**18.what are the different ways the Linux can switch from User Space to Kernel Space & vice-versa ?**

**19. Explain about function pointer**

**20.how many storage classes there? Can u explain static**

**21.can u explain about dynamic memory allocation**

**22.what is memcpy?**

**23.what is the difference between kmalloc& Vmalloc**

**24.what is highmem?**

**25.when will u use ISR's?**

**26.what is structure padding?**

. What is the difference in features between kernel 2.2, 2.4 and 2.6 ?

2. What are Static and Shared libraries ?

3. What is dynamic linking ? What is static linking ?
4. What are the advantages of Dynamic linking or Shared libraries ?
5. Does gcc search for both static and shared libraries ? Which is searched initially by gcc compiler ?
6. What should be done for Shared library based linking in gcc ?
7. What should be done for static library based linking in gcc ?
8. What is object file and what are symbols ?
9. Can you tell the memory layout based on Data,BSS,HEAP and STACK ?
10. What are the ways in which linux kernel can be compiled ?
11. How will get the driver added into the kernel ? What are Kconfig files ?
12. What is a kernel module ?
13. What is the difference between insmod and modprobe ?
14. How will you list the modules ?
15. How do you get the list of currently available drivers ?
16. How will you Access userspace memory from kernel ? What are the various methods ?
17. What is the use of ioctl(inode,file,cmd,arg) API ?
18. What is the use of the poll(file, polltable) API ?
19. What is the use of file->private\_data in a device driver structure ?
20. What is a device number ?
21. What are the two types of devices drivers from VFS point of view ?
22. What are character devices ?
23. How does the character device driver adds and remove itself from the kernel ?  
What is the use of register\_chrdev and unregister\_chrdev ?
24. What is the role of interrupts in a device driver ? How are interrupts handled in device driver ?
25. How will you make interrupt handlers as fast as possible ?
26. What are the types of softirqs ?
27. Difference between Timer Softirq and Tasklet Softirq ?
28. What are tasklets ? How are they activated ? when and How are they initialized ?
29. What is task\_struct and how are task states maintained ?
30. What is rwlock and spinlock ? Briefly explain about both of them ?
31. When will you use rwlock instead of spinlock ?
32. Can spinlock/rwlock be used in Interrupt handler ?
33. Tell about the Memory Layout of a Process in Linux .
34. How will you trace the system calls made into the kernel of lInux ?
35. What is mmap ? MMAP & malloc ? MMAP & brk ? MMAP adv & dis-adv.
36. Tell the relation between Malloc and MMAP
37. Advantages of MMAP over Read ?
38. Tell the role of brk() in malloc / Tell the relation between heap and brk?
39. Example of using MMAP and MUNMAP in C ?
40. Tell about the method/steps in Linux Kernel Compilation.
41. What is Kmalloc and how does it differ from normal malloc ? or Why can't we use malloc in kernel code ?
42. What happens as soon as a packet arrives from the network in Linux ?
43. What is a stack frame, stack pointer & frame pointer ?
44. What is a profiler ? Which one have you used ?
45. How do you determine the direction of stack growth ?

Describe initial process sequence while the system boots up.

1) BIOS 2) Master Boot Record (MBR) 3) LILO or GRUB 4) Kernel 5) init 6) Run Levels.....

What is a shell? What are shell variables?

A shell us an interface to the user of any operating system.....

Explain how the inode maps to data block of a file.

There are 13 block addresses in inode. The file descriptions – type of file, access rights.....

Explain some system calls used for process management.

The following are the system calls: fork() - For creating child process .....

Explain how to get/set an environment variable from a program.

An environment variable can get using the function getenv().....

Describe how a parent and child process communicates each other.

The inter communication between a child process and a parent process can be done through normal communication.....

What is a Daemon?

Daemon is the short form for Disk and Execution Monitor.....

What is 'ps' command for?

The shortage for “process status” is ps. This command is used to display the currently running processes on Linux/Unix systems.....

How the Swapper works?

Moving the information from fast access memory and slow access memory and vice versa is known as swapping.....

What is the difference between Swapping and Paging?

Swapping performs the whole process to transfer to the disk, where as paging performs.....

What is Expansion swap?

Expansion swap is a part of hard disk. This is reserved for the purpose of storing chunks of a program.....

### What is Fork swap?

For creation of child process, fork() system call is invoked. At the time of processing the fork() call by parent.....

### What are the requirements for a swapper to work?

The functionality of a swapper is on the scheduling priority which is highest. The swapper searches.....

### What is 'the principle of locality'?

The next most data item or instruction is the closest to the current data item or instruction.....

### What is page fault? Its types.

One of the critical parts of code in the Linux kernel. It has a major influence on memory subsystem's performance.....

### Difference between the fork() and vfork() system call.

fork: Both the parent and child share all of the page tables until any one of them does a write.....

### What is BSS(Block Started by Symbol)?

UNIX linkers produce uninitialized data segments.....

### What is Page-Stealer process? Explain the paging states for a page in memory.

The pages that are eligible for swapping are found by the Page-Stealer' .....

### Explain the phases of swapping a page from the memory.

The phases of swapping a page from the memory are:.....

### What is Demand Paging? Explain the conditions for a machine to support Demand Paging.

The process of mapping a large address space into a relatively small amount of physical memory is known as demand paging.....

### Difference between Fault Handlers and the Interrupt handlers.

Fault handlers can sleep, where as interrupt handlers cannot.....

### What is validity fault? In what way the validity fault handler concludes?

Validity fault is the result of non setting of valid bits in main memory at the time of referring a page by a process .....

### What is protection fault?

Protection fault is a name of an error. This error occurs when accessing storage space is tried.....

Explain how the Kernel handles both the page stealer and the fault handler.

When the memory shortage occurs then the page stealer and fault handler thrashes.....

What is ex and vi? Explain their purposes.

ex – the line editor mode of ‘vi’ editor. It allows to.....

What is kill()? Explain its possible return values.

kill() is a system call which stops a process. The return values of kill() are:.....

Explain the steps that a shell follows while processing a command.

The sequence of executing commands by shell are as follows:.....

What is the difference between cmp and diff commands? Provide an example for each.

Byte by byte comparision performed for two files comparision and displays the first mismatch byte.....

What is the use of ‘grep’ command? Provide an example

Grep stands for regular expression. ‘grep’ command is used to find the patterns in a text file provided by the user.....

Difference between cat and more command.

The file contents are displayed by ‘cat’ command.....

What is ‘du’ command? What is its use?

The du (disk usage) command is used to report the size of directory trees.....

Explain the various prompts that are available in a UNIX system.

UNIX supports 4 prompts: PS1: default prompt.....

Describe how the kernel differentiates device files and ordinary files.

There are 2 device files. They are character device file and block device file.....

Explain how to switch to a super user status to gain privileges.

The command ?su? is used to get super user status.....

Linux - What are Pipes? Explain use of pipes

[NEXT>>](#)

*Linux - What are Pipes? Explain use of pipes - August 21, 2008 at 22:00 pm by Rajmeet Ghai*

What are Pipes? Explain use of pipes.

### **Answer**

A pipe is a chain of processes so that output of one process (stdout) is fed an input (stdin) to another. UNIX shell has a special syntax for creation of pipelines. The commands are written in sequence separated by |. Different filters are used for Pipes like AWK, GREP.

e.g. sort file | lpr ( sort the file and send it to printer)

Uses of Pipe

Several powerful functions can be in a single statement

Streams of processes can be redirected to user specified locations using >

*Linux - What are Pipes? Explain use of pipes - August 21, 2008 at 22:00 pm by Rajmeet Ghai*

What are Pipes? Explain use of pipes.

Pipe is a symbol used to provide output of one command as input to another command. The output of the command to the left of the pipe is sent as input to the command to the right of the pipe. The symbol is |.

For example:

\$ cat apple.txt | wc

In the above example the output of apple.txt file will be sent as input for wc command which counts the no. of words in a file. The file for which the no. of words counts is the file apple.txt.

Pipes are useful to chain up several programs, so that multiple commands can execute at once without using a shell script

Linux - What is Linux and why is it so popular?

[NEXT>>](#)

*Linux - What is Linux and why is it so popular? - August 21, 2008 at 22:00 pm by Rajmeet Ghai*

What is Linux and why is it so popular?

### **Answer**

Linux is an operating system that uses UNIX like Operating system. However, unlike UNIX, Linux is an open source and free software. Linux was originally created by Linus Torvalds and commonly used in servers.

Popularity of Linux is because of the following reasons

- It is free and open source. We can download Linux for free and customize it as per our needs.
- It is very robust and adaptable.
- Immense amount of libraries and utilities

*Linux - What is Linux and why is it so popular? - May 11, 2009 at 13:00 pm by Vidya Sagar*

*What is Linux and why is it so popular?*

Linux is a multiuser, multitask GUI based open source operating system developed by Linus Torvalds. Torvalds has invited the community to enhance the Linux kernel and thousands of system programmers worked on to enhance.

Prior to Linux, there is UNIX. The desktop work stations from various companies were based on UNIX. Later a numerous companies entered and each one of them had their own UNIX version. As the proprietary authority is owned by each company and the lack of central authority weaken UNIX. As Linux is free and runs on any PC platform it gained the popularity very quickly. The following are few more reasons for its popularity:

- People who are familiar with UNIX can work on Linux with ease and comfort.
- People who want great control over network security and on operating system

How does the linux file system work?

**Answer**

Linux file structure is a tree like structure. It starts from the root directory, represented by '/', and then expands into sub-directories. All the partitions are under the root directory. If a partition is mounted (The mount point defines the place of a particular data set in the file system) anywhere apart from a "device", the system is not aware of the existence of that partition or device. Directories that are only one level below the root directory are often preceded by a slash, to indicate their position.

Explain file system of linux. The root "/" filesystem, /usr filesystem, /var filesystem, /home filesystem, /proc filesystem.

**Answer**

**Root "/" file system:** The kernel needs a root file system to mount at start up. The root file system is generally small and should not be changed often as it may interrupt in booting. The root directory usually does not have the critical files. Instead sub directories are created. E.g. /bin (commands needed during bootup), /etc (config files) , /lib(shared libraries).

**/usr filesystem :** this file system is generally large as it contains the executable files to be shared amongst different machines. Files are usually the ones installed while installing Linux. This makes it possible to update the system from a new version of the distribution, or even a completely new distribution, without having to install all programs again. Sub directories include /bin, /include, /lib, /local (for local executables)

**/var filesystem :** this file system is specific to local systems. It is called as var because the data keeps changing. The sub directories include /cache/man (A cache for man pages), /games (any variable data belong to games), /lib (files that change), /log (log from different programs), /tmp (for temporary files)

**/home filesystem:** - this file system differs from host to host. User specific configuration files for applications are stored in the user's home directory in a file. UNIX creates directories for all users directory. E.g /home/my\_name. Once the user is logged in ; he is placed in his home directory.

**/proc filesystem :** this file system does not exist on the hard disk. It is created by the kernel in its memory to provide information about the system. This information is usually about the processes. Contains a hierarchy of special files which represent the current state of the kernel .Few of the Directories include /1 (directory with information about process num 1, where 1 is the identification number), /cpuinfo (information about cpu), /devices (information about devices installed), /filesystem (file systems configured), /net (information about network protocols), /mem (memory usage)

What is Kernel? Explain the task it performs.

### **Answer**

Kernel is used in UNIX like systems and is considered to be the heart of the operating system. It is responsible for communication between hardware and software components. It is primarily used for managing the systems resources as well.

Kernel Activities:

The Kernel task manager allows tasks to run concurrently.

Managing the computer resources: Kernel allows the other programs to run and use the resources. Resources include i/o devices, CPU, memory.

Kernel is responsible for Process management. It allows multiple processes to run simultaneously allowing user to multitask.

Kernel has an access to the systems memory and allows the processes to access the memory when required. Processes may also need to access the devices attached to the system. Kernel assists the processes in doing so.

For the processes to access and make use of these services, system calls are used.

*What is Kernel? Explain the task it performs.*

Kernel is the component that is responsible for managing the resources of a computer system.

The tasks are:

- Provides the abstraction level for resources such as memory, processors, and I/O devices.
- Performs inter process communication
- Responds to system calls
- Provides methods for synchronization and communication between processes.

Explain each system calls used for process management in linux.

### Answer

System calls used for Process management:

Fork () :- Used to create a new process  
Exec() :- Execute a new program  
Wait():- wait until the process finishes execution  
Exit():- Exit from the process  
Getpid():- get the unique process id of the process  
Getppid():- get the parent process unique id  
Nice():- to bias the existing property of process

*Explain each system calls used for process management in linux.*

Process management uses certain system calls. They are explained below.

1. To create a new process – fork () is used.
2. To run a new program = exec () is used.
3. To make the process to wait = wait () is used.
4. To terminate the process – exit () is used.
5. To find the unique process id – getpid () is used.
6. To find the parent process id – getppid () is used.
7. To bias the currently running process property – nice () is used.

What are the process states in Linux?

### Answer

Process states in Linux:

**Running:** Process is either running or ready to run  
**Interruptible:** a Blocked state of a process and waiting for an event or signal from another process  
**Uninterruptible:** a blocked state. Process waits for a hardware condition and cannot handle any signal  
**Stopped:** Process is stopped or halted and can be restarted by some other process  
**Zombie:** process terminated, but information is still there in the process table.

*What are the process states in Linux?*

The following are the process states:

1. **Running:** This is a state where a process is either in running or ready to run.
2. **Interruptible:** This state is a blocked state of a process which awaits for an event or a signal from another process
3. **Uninterruptible:** It is also a blocked state. The process is forced to halt for certain condition that a hardware status is waited and a signal could not be handled.
4. **Stopped:** Once the process is completed, this state occurs. This process can be restarted
5. **Zombie:** In this state, the process will be terminated and the information will still be available in the process table.

### What is Linux Shell? What is Shell Script?

#### Answer

Linux shell is a user interface used for executing the commands. Shell is a program the user uses for executing the commands. In UNIX, any program can be the users shell. Shell categories in Linux are:

Bourne shell compatible, C shell compatible, nontraditional, and historical.

A shell script, as the name suggests, is a script written for the shell. Script here means a programming language used to control the application. The shell script allows different commands entered in the shell to be executed. Shell script is easy to debug, quicker as compared to writing big programs. However the execution speed is slow because it launches a new process for every shell command executed. Examples of commands are cp, cn, cd.

*Linux - What is Linux Shell? What is Shell Script? - August 21, 2008 at 22:00 pm by Rajmeet Ghai  
What is Linux Shell? What is Shell Script?*

Linux shell is the user interface to communicate with Linux operating system. Shell interprets the user requests, executes them. Shell may use kernel to execute certain programs. Shell Script: A shell script is a program file in which certain Linux commands are placed to execute one after another. A shell script is a flat text file. Shell scripts are useful to accept inputs and provide output to the user. Everyday automation process can be simplified by a shell script.

.....

### Mindlance Interview Questions: For Post Silicon Validation

#### First Round: Written test

- 1) Half Adder and Full Adder Circuit
- 2) Pythagoras Theorem
- 3) Lab View Program Example
- 4) Verilog Code Example
- 5) About UART,ADC,PCB,IOT,Data Structures,Data Types
- 6) Different Types of Post Silicon Validation
- 7) Algebraic Equations
- 8) Flip Flop Concepts
- 9) Boolean Expressions for circuits
- 10) What is Post and Pre-Silicon validation
- 11) Difference Between Synchronous and asynchronous Reset
- 12) Interrupt Latency

#### Second Round: F2F

- 1) Tell me about Yourself
  - 2) Resume Discussion
  - 3) Asked about Verilog and lab view experience
- .....
- 4) Delphi Automotive(9th may 2014)**

- 5) 1.Explain IIC
- 6) 2)How many stop bits does IIC frame will have
- 7) 3)Write macro for multiplication of 2 numbers
- 8) 4) How to shift array elements in towards left
- 9) 5) How to print array in reverse order

**10) HCL(10th may 2014) (PERL)**

- 11) 1) Write a pattern to match a valid mail ID
- 12) 2)Stop execution of the program when certain condition satisfies
- 13) 3)How to read a file in random manner and a particular line
- 14) 4)Explain where PERL is used in your project flow.
- 15) 5)How to get the present time
- 16) 6)Difference between my , local, our keywords
- 17) 7) When memory is allocated for a variable
- 18) 8) How to pass a array as reference to subroutine

**19) SAMSUNG(9th June) [ Firmware ]**

- 20) 1) Explain IIC
- 21) 2) If 3 devices have same address and there is no A0,A1,A2 facility then how master will differentiate 3 slaves and which device responds
- 22) 3) In UART explain how handshaking signals work
- 23) 4) what type of UART u have worked on , pooling based or interrupt based.
- 24) 5) What are the worst cases where data lost in polling based method
- 25) 6) What is watch dog timer
- 26) 7) Explain 2\*16 LCD and what is the size of RAM you are using
- 27) 8) Explain how RTC works
- 28) 9) Explain malloc and is it necessary to do type casting , without what is the result
- 29) 10) volatile type qualifier and where these variables are stored in memory
- 30) 11) void pointer
- 31) 12) Range of values stored by using 8 bit format
- 32) 13) suppose signed char b=-4; how this will be stored in memory

**33) Robert Bosch (24th June) [ C & C++ ]**

- 34) 1) Storage Classes
- 35) 2) When size will be allocated in the structure
- 36) 3) Difference between structure and Union
- 37) 4) Padding in structure
- 38) 5) Difference between C structures and C structures
- 39) 6) Explain OOPS concepts
- 40) 7) Function Overloading
- 41) 8) How u write functions with variable number of parameters (variadic Functions)
- 42) 9) Process and threads

43) 10) static and global variables

44) 11) Multithreaded Programs

45) 12) Swap 2 variables with out using 3 rd variable

46) 13) Reversing a string with out array or bit variable or buffer

### **47) Robert Bosch (27th Sep) [ Perl & Testing ]**

48) 1. WAP to read contents of file1 , file2 compare the stings in them and print all common to one file and which are not common to other files .

49) 2. Testing Concepts

50) 3. Difference between Linux , windows OS

51) 4.Difference between black box and white box testing

52) 5. what is System testing.

53) 6. BE project

### **54) Hubbell (C & Embedded) 10th oct 2014**

55) 1. Tell me something about your self

56) 2. Projects , Questions on Projects with respect to embedded concepts and protocols , like IC numbers , clock frequency etc...

57) 3. Compiler u have used , frame work

58) 4. What is inode number

59) 5. Factors influencing selection of microcontroller

60) 6. Structure padding

61) 7. How to toggle bits .

62) 8. Explain IIC , UART w r t project , why IIC not other protocols

### **63)**

### **64) Robert Bosch (C & Embedded) 11th Oct 2014**

#### **65) 1 st Round**

66) 1. Tell me something about yourself

67) 2. Project , block diagram , your role , interfacing h/w , specifications of h/w ,

68) 3. How to measure UART speed , Baud rate

69) 4. IIC Protocol , asked me to explain procedure for starting communication , start bit etc..

70) 5. How to find the number is even or odd without using arithmetic operators , address operator , Unary operators

#### **71) 2nd Round**

72) 1 . Difference between Processor and micro controller

73) 2. Have u got any awards/ rewards when working with your last employer

74) 3. Why you are willing to change your job

75) 4. Future aims , promise me how long u are going to stay with us

76) 5. Do you have any automotive experience & few more

#### **77) 3rd Round (HR ) - 21st Oct**

78) 1. Tell me some thing about yourself (Background , Educational , professional )

- 79) 2. Why you are willing to change your job
- 80) 3. What inspires you to join BOSCH .
- 81) 4. why you quit your previous company .
- 82) 5. Key technical competency area
- 83) 6 . Notice Period , package etc ...
- 84)

**85) Continental Automotive(24th Oct 2014)**

**86) 1st Round**

- 87) 1. Explain How UART works internally , baud rate
- 88) 2. How SPI works , frame format
- 89) 3. Declaring a function which returns more than 2 values , function pointer for it .
- 90) 4. storage classes
- 91) 5. pattern matching
- 92) 6. how match a number in number format & alphabet format .
- 93) 7. Explain the types of memories available .

**94) 2nd Round**

- 95) 1. Projects ,
- 96) 2. Difference between ! and ~operator
- 97) 3 . Embedded C program which produces a square wave of 50% duty cycle and when off produces a duty cycle of 0 %

**98)**

**99) HR Round**

- 100)
- 101)

---

### Linux Device Model (LDM)

Explain about the Linux Device Model (LDM)?

Explain about ksets, kobjects and ktypes. How are they related?

Questions about sysfs.

### Linux Boot Sequence

Explain about the Linux boot sequence in case of ARM architecture?

How are the command line arguments passed to Linux kernel by the u-boot (bootloader)?

Explain about ATAGS?

Explain about command line arguments that are passed to linux kernel and how/where they are parsed in kernel code?

Explain about device tree.

### Interrupts Interrupts in Linux

Explain about the interrupt mechanisms in linux?

What are the APIs that are used to register an interrupt handler?

How do you register an interrupt handler on a shared IRQ line?

Explain about the flags that are passed to request\_irq().

Explain about the internals of Interrupt handling in case of Linux running on ARM.

What are the precautions to be taken while writing an interrupt handler?

Explain interrupt sequence in detail starting from ARM to registered interrupt handler.

What is bottom half and top half.

What is request\_threaded\_irq()

If same interrupts occurs in two cpu how are they handled?

How to synchronize data between 'two interrupts' and 'interrupts and process'.

How are nested interrupts handled?

How is task context saved during interrupt.

### Bottom-half Mechanisms in Linux

What are the different bottom-half mechanisms in Linux?

Softirq, Tasklet and Workqueues

What are the differences between Softirq/Tasklet and Workqueue? Given an example what you prefer to use?

What are the differences between softirqs and tasklets?

Softirq is guaranteed to run on the CPU it was scheduled on, whereas tasklets don't have that guarantee.

The same tasklet can't run on two separate CPUs at the same time, whereas a softirq can.

When are these bottom halves executed?

Explain about the internal implementation of softirqs?

<http://linuxblore.blogspot.com/2013/02/bottom-halves-in-linux-part-1-softirqs.html>

Explain about the internal implementation of tasklets?

<http://linuxblore.blogspot.com/2013/02/bottom-halves-in-linux-part-2-tasklets.html>

Explain about the internal implementation of workqueues?

<http://linuxblore.blogspot.in/2013/01/workqueues-in-linux.html>

Explain about the concurrent work queues.

I, please click on Like in the below facebook snippet

### Kernel Synchronization

Why do we need synchronization mechanisms in Linux kernel?

What are the different synchronization mechanisms present in Linux kernel?

What are the differences between spinlock and mutex?

What is lockdep?

Which synchronization mechanism is safe to use in interrupt context and why?

Explain about the implementation of spinlock in case of ARM architecture.

Explain about the implementation of mutex in case of ARM architecture.

Explain about the notifier chains.

Explain about RCU locks and when are they used?

Explain about RW spinlocks locks and when are they used?

Which are the synchronization techniques you use 'between processes', 'between process and interrupt' and 'between interrupts'; why and how ?

What are the differences between semaphores and spinlocks?

### Process Management and Process Scheduling

What are the different schedulers class present in the linux kernel?

How to create a new process?

What is the difference between fork( ) and vfork( )?

Which is the first task what is spawned in linux kernel?

What are the processes with PID 0 and PID 1?

PID 0 - idle task

PID 1 - init

How to extract task\_struct of a particular process if the stack pointer is given?

How does scheduler picks particular task?

When does scheduler picks a task?

How is timeout managed?

How does load balancing happens?

Explain about any scheduler class?

Explain about wait queues and how they implemented? Where and how are they used?

What is process kernel stack and process user stack? What is the size of each and how are they allocated?

Why do we need separate kernel stack for each process?

What all happens during context switch?

What is thread\_info? Why is it stored at the end of kernel stack?

What is the use of preempt\_count variable?

What is the difference between interruptible and uninterruptible task states?

How processes and threads are created? (from user level till kernel level)

How is virtual run time (vruntime) calculated?

### Timers and Time Management

What are jiffies and HZ?

What is the initial value of jiffies when the system has started?

Explain about HR timers and normal timers?

On what hardware timers, does the HR timers are based on?

How to declare that a specific hardware timer is used for kernel periodic timer interrupt used by the scheduler?

How software timers are implemented?

### Power Management in Linux

Explain about cpuidle framework.

Explain about cpufreq framework.

Explain about clock framework.

Explain about regulator framework.

Explain about suspended and resume framework.

Explain about early suspend and late resume.

Explain about wakelocks.

### Linux Kernel Modules

How to make a module as loadable module?

How to make a module as in-built module?

Explain about Kconfig build system?

Explain about the init call mechanism.

What is the difference between early init and late init?

Early init:

Early init functions are called when only the boot processor is online.

Run before initializing SMP.

Only for built-in code, not modules.

Late init:

Late init functions are called \_after\_ all the CPUs are online.

### Linux Kernel Debugging

What is Oops and kernel panic?

Does all Oops result in kernel panic?

What are the tools that you have used for debugging the Linux kernel?

What are the log levels in printk?

Can printk's be used in interrupt context?

How to print a stack trace from a particular function?

What's the use of early\_printk( )?

Explain about the various gdb commands.

### Miscellaneous

How are the atomic functions implemented in case of ARM architecture?

How is container\_of( ) macro implemented?

Explain about system call flow in case of ARM Linux.

What 's the use of \_\_init and \_\_exit macros?

How to ensure that init function of a particular driver was called before our driver's init function is called (assume that both these drivers are built into the kernel image)?

What's a segmentation fault and what are the scenarios in which segmentation fault is triggered?

If the scenarios which triggers the segmentation fault has occurred, how the kernel identifies it and what are the actions that the kernel takes?

This page is updated every week with latest questions. Bookmark this and keep watching.

Embedded systems interview questions

Linux kernel and Linux device driver interview questions

C Programming Interview Questions

ARM Interview Questions

Operating System Interview Questions

Boot loader and boot sequence interview questions

Microprocessor and Embedded Hardware Interview Questions

Linux kernel and Linux device driver interview questions

The following are the questions that are asked in various interviews, some of which are listed below:

?

Broadcom interview questions

Qualcomm interview questions

Cisco interview questions

Juniper networks interview questions

AMD interview questions

Intel interview questions

And some other companies

### Linux Device Model (LDM)

Explain about the Linux Device Model (LDM)?

Explain about ksets, kobjects and ktypes. How are they related?

Questions about sysfs.

### Linux Boot Sequence

Explain about the Linux boot sequence in case of ARM architecture?

How are the command line arguments passed to Linux kernel by the u-boot (bootloader)?

Explain about ATAGS?

Explain about command line arguments that are passed to linux kernel and how/where they are parsed in kernel code?

Explain about device tree.

### Interrupts in Linux

Explain about the interrupt mechanisms in linux?

What are the APIs that are used to register an interrupt handler?

How do you register an interrupt handler on a shared IRQ line?

Explain about the flags that are passed to request\_irq().

Explain about the internals of Interrupt handling in case of Linux running on ARM.

What are the precautions to be taken while writing an interrupt handler?

Explain interrupt sequence in detail starting from ARM to registered interrupt handler.

What is bottom half and top half.

What is request\_threaded\_irq()

If same interrupts occurs in two cpu how are they handled?

How to synchronize data between 'two interrupts' and 'interrupts and process'.

How are nested interrupts handled?

How is task context saved during interrupt.

### Bottom-half Mechanisms in Linux

What are the different bottom-half mechanisms in Linux?

Softirq, Tasklet and Workqueues

What are the differences between Softirq/Tasklet and Workqueue? Given an example what you prefer to use?

What are the differences between softirqs and tasklets?

Softirq is guaranteed to run on the CPU it was scheduled on, whereas tasklets don't have that guarantee.

The same tasklet can't run on two separate CPUs at the same time, whereas a softirq can.

When are these bottom halves executed?

Explain about the internal implementation of softirqs?

<http://linuxblore.blogspot.com/2013/02/bottom-halves-in-linux-part-1-softirqs.html>

Explain about the internal implementation of tasklets?

<http://linuxblore.blogspot.com/2013/02/bottom-halves-in-linux-part-2-tasklets.html>

Explain about the internal implementation of workqueues?

<http://linuxblore.blogspot.in/2013/01/workqueues-in-linux.html>

Explain about the concurrent work queues.

\*\*\*\*\*

*Define the scope of static variables.*

The scope of a static variable is local to the block in which the variable is defined. However, the value of the static variable persists between two function calls.

**C - scope of static variables - Jan 11, 2010 at 14:55 PM by Vidyasagar**

*Define the scope of static variables.*

Static variables in C have the scopes;

1. Static global variables declared at the top level of the C source file have the scope that they can not be visible external to the source file. The scope is limited to that file.
2. Static local variables declared within a function or a block, also known as local static variables, have the scope that, they are visible only within the block or function like local variables. The values assigned by the functions into static local variables during the first call of the function will persist / present / available until the function is invoked again.

The static variables are available to the program, not only for the function / block. It has the scope within the current compile. When static variable is declared in a function, the value of the variable is preserved , so that successive calls to that function can use the latest updated value. The static variables are initialized at compile time and kept in the executable file itself. The life time extends across the complete run of the program.

Static local variables have local scope. The difference is, storage duration. The values put into the local static variables, will still be present, when the function is invoked next time.

### *What are volatile variables?*

Volatile variables get special attention from the compiler. A variable declared with the volatile keyword may be modified externally from the declaring function.

If the keyword volatile is not used, the compiler optimization algorithms might consider this to be a case of infinite loop. Declaring a variable volatile indicates to a compiler that there could be external processes that could possibly alter the value of that variable.

e.g.:

A variable that might be concurrently modified by multiple threads may be declared volatile. Variables declared to be volatile will not be optimized by the compiler. Compiler must assume that their values can change at any time. However, operations on a volatile variable are still not guaranteed to be atomic.

### **C - What are volatile variables? - Jan 11, 2010 at 12:44 PM by Vidya Sagar**

### *What are volatile variables?*

Volatile variables are like other variables except that the values might be changed at any given point of time only by 'some external resources'.

Ex:

```
volatile int number;
```

The value may be altered by some external factor, though if it does not appear to the left of the assignment statement. The compiler will keep track of these volatile variables

### *Explain the meaning of "Segmentation violation".*

A segmentation violation usually indicates an attempt to access memory which doesn't even exist.

### **C - meaning of "Segmentation violation" - Jan 11, 2010 at 12:15 PM by Vidya Sagar**

### *Explain the meaning of "Segmentation violation".*

Segmentation violation usually occurs at the time of a program's attempt for accessing memory location, which is not allowed to access. The following code should create segmentation violation.

```
main() {
char *hello = "Hello World";
*hello = 'h';
}
```

At the time of compiling the program, the string “hello world” is placed in the binary mark of the program which is read-only marked. When loading, the compiler places the string along with other constants in the read-only segment of the memory. While executing a variable \*hello is set to point the character ‘h’ , is attempted to write the character ‘h’ into the memory, which cause the segmentation violation. And, compiler does not check for assigning read only locations at compile time.

### *What does static variable mean in C?*

Static variable is available to a C application, throughout the life time. At the time of starting the program execution, static variables allocations takes place first. In a scenario where one variable is to be used by all the functions (which is accessed by main () function), then the variable need to be declared as static in a C program. The value of the variable is persisted between successive calls to functions. One more significant feature of static variable is that, the address of the variable can be passed to modules and functions which are not in the same C file.

### *What are bitwise shift operators?*

The bitwise operators are used for shifting the bits of the first operand left or right. The number of shifts is specified by the second operator.

Expression << or >> number of shifts

Ex:

```
number<<3; /* number is an operand - shifts 3 bits towards left*/
number>>2; /* number is an operand – shifts 2 bits towards right*/
```

The variable number must be an integer value.

For leftward shifts, the right bits those are vacated are set to 0. For rightward shifts, the left bits those are vacated are filled with 0’s based on the type of the first operand after conversion.

If the value of ‘number’ is 5, the first statement in the above example results 40 and stored in the variable ‘number’.

If the value of ‘number’ is 5, the second statement in the above example results 0 (zero) and stored in the variable ‘number’.

### **C - bitwise shift operators - August 06, 2008 at 13:10 PM by Amit Satpute**

#### *What are bitwise shift operators?*

##### **<< - Bitwise Left-Shift**

Bitwise Left-Shift is useful when to want to MULTIPLY an integer (not floating point numbers) by a power of 2.

Expression: a << b

This expression returns the value of a multiplied by 2 to the power of b.

##### **>> - Bitwise Right-Shift**

Bitwise Right-Shift does the opposite, and takes away bits on the right.

Expression: a >> b

This expression returns the value of a divided by 2 to the power of b.

#### *Explain the use of bit field.*

Packing of data in a structured format is allowed by using bit fields. When the memory is a premium, bit fields are extremely useful. For example:

- Picking multiple objects into a machine word : 1 bit flags can be compacted
- Reading external file formats : non-standard file formats could be read in, like 9 bit integers

This type of operations is supported in C language. This is achieved by putting ':bit length' after the variable. Example:

```
struct packed_struct {
 unsigned int var1:1;
 unsigned int var2:1;
 unsigned int var3:1;
 unsigned int var4:1;
 unsigned int var5:4;
 unsigned int funny_int:9;
} pack;
```

packed-struct has 6 members: four of 1 bit flags each, and 1 4 bit type and 1 9 bit funny\_int.

C packs the bit fields in the structure automatically, as compactly as possible, which provides the maximum length of the field is less than or equal to the integer word length the computer system.

The following points need to be noted while working with bit fields:

- The conversion of bit fields is always integer type for computation
- Normal types and bit fields could be mixed / combined
- Unsigned definitions are important.

*What is the purpose of "register" keyword?*

It is used to make the computation faster.

The register keyword tells the compiler to store the variable onto the CPU register if space on the register is available. However, this is a very old technique. Todays processors are smart enough to assign the registers themselves and hence using the register keyword can actually slowdown the operations if the usage is incorrect.

**C - purpose of "register" keyword - Jan 11, 2010 at 12:55 PM by Vidya Sagar**

*What is the purpose of "register" keyword?*

The keyword 'register' instructs the compiler to persist the variable that is being declared , in a CPU register.

Ex: register int number;

The persistence of register variables in CPU register is to optimize the access. Even the optimization is turned off; the register variables are forced to store in CPU register.

*Explain the use of "auto" keyword.*

When a certain variable is declared with the keyword 'auto' and initialized to a certain value, then within the scope of the variable, it is reinitialized upon being called repeatedly.

**C - use of "auto" keyword - Jan 11, 2010 at 14:50 PM by Vidya Sagar**

*Explain the use of "auto" keyword.*

The keyword 'auto' is used extremely rare. A variable is set by default to auto. The declarations:

auto int number; and int number;

has the same meaning. The variables of type static, extern and register need to be explicitly declared, as their need is very specific. The variables other than the above three are implied to be of 'auto' variables. For better readability auto keyword can be used.

*Compare array with pointer.*

The following declarations are NOT the same:

```
char *p;
char a[20];
```

The first declaration allocates memory for a pointer; the second allocates memory for 20 characters.

**C - Compare array with pointer - Jan 11, 2010 at 18:10 PM by Vidya Sagar**

*Define pointer in C.*

A pointer is an address location of another variable. It is a value that designates the address or memory location of some other value (usually value of a variable). The value of a variable can be accessed by a pointer which points to that variable. To do so, the reference operator (&) is pre-appended to the variable that holds the value. A pointer can hold any data type, including functions.

*What is NULL pointer?*

A null pointer does not point to any object.

NULL and 0 are interchangeable in pointer contexts. Usage of NULL should be considered a gentle reminder that a pointer is involved.

It is only in pointer contexts that NULL and 0 are equivalent. NULL should not be used when another kind of 0 is required.

**C - What is NULL pointer? - Jan 11, 2010 at 18:10 PM by Vidya Sagar**

*What is NULL pointer?*

A pointer that points to a no valid location is known as null pointer. Null pointers are useful to indicate special cases. For example, no next node pointer in case of a linked list. An indication of errors that pointers returned from functions.

## C question answer

#####

Local Variables are stored in Stack. Register variables are stored in Register. Global & static variables are stored in data segment. The memory created dynamically are stored in Heap And the C program instructions get stored in code segment and the extern variables also stored in data segment.

constants are often left in ROM

#####

### **Static Variable-**

There are 3 main uses for the static.

1. If you declare within a function: It retains the value between function calls
2. If it is declared for a function name: By default function is extern..so it will be visible from other files if the function declaration is as static..it is invisible for the outer files
3. Static for global variables:

By default we can use the global variables from outside files If it is static global..that variable is limited to with in the file.

if static variable declared in class-

static variables are those variables whose values are shared among various instance of an class. For e.g. if you have a static variable "x" in class "A" and you create two instances of A i.e. a1 and a2. In that case a1 and a2 will share the common variable. This means if a2 changes the value of x than this will be changed for a1 as well.

Class A

```
{
static int x;
}
```

```
A a1=new A();
A a2=new A();
a1.x=10;
then a2.x will also become 10.
```

\*\* Default initial value of static integral type variables are zero.

\*If declared a static variable or function globally then its visibility will only the file in which it has declared

#####

### **What are the differences between structures and arrays?**

Ans: Structure is a collection of heterogeneous data type but array is a collection of homogeneous data types.

#### **Array**

- 1-It is a collection of data items of same data type.
- 2-It has declaration only
- 3-.There is no keyword.
- 4- array name represent the address of the starting element.

#### **Structure**

- 1-It is a collection of data items of different data type.
- 2- It has declaration and definition
- 3- keyword struct is used
- 4-Structure name is known as tag it is the short hand notation of the declaration.

#####

#### **Union-**

A **union** is a special data type available in C that enables you to store different data types in the same memory location. You can define a union with many members, but only one member can contain a value at any given time.

Unions provide an efficient way of using the same memory location for multi-purpose.

```
union Data
{
 int i;
 float f;
 char str[20];
} data
```

Now a variable of **Data** type can store an integer, a floating-point number, or a string of characters. This means that a single variable ie. same memory location can be used to store multiple types of data. You can use any built-in or user defined data types inside a union based on your requirement.

The memory occupied by a union will be large enough to hold the largest member of the union. For example, in above example **Data** type will occupy 20 bytes of memory space because this is the maximum space which can be occupied by character string. Following is the example which will display total memory size occupied by the above union:

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

union Data
{
 int i;
 float f;
 char str[20];
};

int main()
{
 union Data data;

 data.i = 10;
 data.f = 220.5;
 strcpy(data.str, "C Programming");

 printf("data.i : %d\n", data.i);
 printf("data.f : %f\n", data.f);
 printf("data.str : %s\n", data.str);

 return 0;
}
```

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

```
data.i : 1917853763
data.f : 4122360580327794860452759994368.000000
data.str : C Programming
```

```
#####
```

### **What is a structure?**

Ans: Structure constitutes a super data type which represents several different data types in a single unit. A structure can be initialized if it is static or global

```
#####
```

### **What are the differences between structures and union?**

Ans: A structure variable contains each of the named members, and its size is large enough to hold all the members. Structure elements are of same size.

A union contains one of the named members at a given time and is large enough to hold the largest member. Union element can be of different sizes.

```
#####
```

### **What are the differences between structures and arrays?**

Ans: Structure is a collection of heterogeneous data type but array is a collection of homogeneous data types.

#### **Array**

- 1-It is a collection of data items of same data type.
- 2-It has declaration only
- 3-There is no keyword.
- 4- array name represent the address of the starting element.

#### **Structure**

- 1-It is a collection of data items of different data type.
- 2- It has declaration and definition
- 3- keyword struct is used
- 4-Structure name is known as tag it is the short hand notation of the declaration.

```
#####
```

### **In header files whether functions are declared or defined?**

Ans: Functions are declared within header file. That is function prototypes exist in a header file,not function bodies

```
#####
```

### malloc, realloc, calloc and free

Malloc- Allocates a block of *size* bytes of memory, returning a pointer to the beginning of the block.

```
/* allocate node */
struct node* new_node =(struct node*) malloc(sizeof(struct node));
```

Calloc-

Calloc takes two arguments Calloc(*b,c*) where *b* no of object and *c* size of object

```
void* calloc (size_t num, size_t size);
#####
#####
```

#### **What are the differences between malloc () and calloc ()?**

Ans: Malloc Calloc 1-Malloc takes one argument Malloc(*a*);where a number of bytes 2-memory allocated contains garbage values

1-Calloc takes two arguments Calloc(*b,c*) where *b* no of object and *c* size of object

2-It initializes the contains of block of memory to zerosMalloc takes one argument, memory allocated contains garbage values.

It allocates contiguous memory locations. Calloc takes two arguments, memory allocated contains all zeros, and the memory allocated is not contiguous.

```
#####
#####
```

#### **Realloc-**

```
void* realloc (void* ptr, size_t size);
```

Changes the size of the memory block pointed to by *ptr*.

*ptr*-Pointer to a memory block previously allocated with malloc, calloc or realloc.

Alternatively, this can be a *null pointer*, in which case a new block is allocated (as if malloc was called).

*size*

New size for the memory block, in bytes.

size\_t is an unsigned integral type.

A pointer to the reallocated memory block, which may be either the same as *ptr* or a new location.

There are several possible outcomes with realloc():

- If sufficient space exists to expand the memory block pointed to by *ptr*, the additional memory is allocated and the function returns *ptr*.
- If sufficient space does not exist to expand the current block in its current location, a new block of the size for *size* is allocated, and existing data is copied from the old block to the beginning

- of the new block. The old block is freed, and the function returns a pointer to the new block.
- If the ptr argument is NULL, the function acts like malloc(), allocating a block of size bytes and returning a pointer to it.
- If the argument size is 0, the memory that ptr points to is freed, and the function returns NULL.
- If memory is insufficient for the reallocation (either expanding the old block or allocating a new one), the function returns NULL, and the original block is unchanged.

```
#####
#####
```

### **Free( ) -**

void free (void\* ptr);

### **Deallocate memory block**

A block of memory previously allocated by a call to malloc, calloc or realloc is deallocated, making it available again for further allocations.

If *ptr* does not point to a block of memory allocated with the above functions, it causes *undefined behavior*.

```
#####
#####
```

### **--What are macros? What are its advantages and disadvantages?**

**Definition:** In C and C++, a Macro is a piece of text that is expanded by the preprocessor part of the compiler. This is used in to expand text before compiling.

```
#define VALUE 10
```

Everywhere that VALUE is used the number of 10 will be used instead.

```
int fred[VALUE];
```

This declares an array of 10 ints.

Macros are abbreviations for lengthy and frequently used statements. When a macro is called the entire code is substituted by a single line though the macro definition is of several lines.

The advantage of macro is that it reduces the time taken for control transfer as in case of function.

The disadvantage of it is here the entire code is substituted so the program becomes lengthy if a macro is called several times.

```
#####
#####
```

**Difference between pass by reference and pass by value?**

**Ans:** Pass by reference passes a pointer to the value. This allows the callee to modify the variable directly. Pass by value gives a copy of the value to the callee. This allows the callee to modify the value without modifying the variable. (In other words, the callee simply cannot modify the variable, since it lacks a reference to it.)

```
#####
#####
```

**Difference between arrays and linked list?**

```
#####
#####
```

**enumerations-** An *enumeration* is a data type consisting of a set of named values that represent integral constants, known as *enumeration constants*Eg:

```
enum grain { oats, wheat, barley, corn, rice };
/* 0 1 2 3 4 */

enum grain { oats=1, wheat, barley, corn, rice };
/* 1 2 3 4 5 */

enum grain { oats, wheat=10, barley, corn=20, rice };
/* 0 10 11 20 21 */
```

```
#####
#####
```

**\*\* What are register variables? What are the advantages of using register variables?**

**Ans:** If a variable is declared with a register storage class, it is known as register variable. The register variable is stored in the cpu register instead of main memory. Frequently used variables are declared as register variable as its access time is faster.

Registers are faster than memory to access, so the variables which are most frequently used in a C program can be put in registers using *register* keyword. The keyword *register* hints to compiler that a given variable can be put in a register.

It's compiler's choice to put it in a register or not. Generally, compilers themselves do optimizations and put the variables in register.

- 1) If you use & operator with a register variable then compiler may give an error or warning (depending upon the compiler you are using), because when we say a variable is a register, it may be stored in a register instead of memory and accessing address of a register is invalid

```
int main()
{
 register int i = 10;

 int *a = &i;

 printf("%d", *a);

 getchar();
```

```

 return 0;
}

```

2) *register* keyword can be used with pointer variables. Obviously, a register can have address of a memory location. There would not be any problem with the below program.

```

int main()
{
 int i = 10;
 register int *a = &i;
 printf("%d", *a);
 getchar();
 return 0;
}

```

3) Register is a storage class, and C doesn't allow multiple storage class specifiers for a variable. So, *register* can not be used with *static*. Try below program.

```

int main()
{
 int i = 10;
 register static int *a = &i;
 printf("%d", *a);
 getchar();
 return 0;
}

```

4) There is no limit on number of register variables in a C program, but the point is compiler may put some variables in register and some not.

```
#####
#####
```

### Typedef-

a new name given to the existing data type may be easier to understand the code.

```

typedef struct {
 int scruples;
 int drams;
 int grains;
} WEIGHT;

```

The structure WEIGHT can then be used in the following declarations:

```
WEIGHT chicken, cow, horse, whale;
```

```
#####
#####
```

### typedef vs #define

The typedef interpretation is performed by the compiler whereas #define statements are processed by the pre-processor

Read variable field width in a scanf() from danis ritchie.  
fgets() and gets()

#####

What is recursion?

#####

Differentiate between for loop and a while loop? What are they used for?

Ans: For executing a set of statements fixed number of times we use for loop while when the number of iterations to be performed is not known in advance we use while loop.

#####

Near, Far and huge pointer-

Near, far, and huge pointers aren't part of standard C; they are/were an extension put in by several vendors to deal with segmented memory architectures

All of the stuff in this answer is relevant only to the old 8086 and 80286 segmented memory model.

near: a 16 bit pointer that can address any byte in a 64k segment

far: a 32 bit pointer that contains a segment and an offset. Note that because segments can overlap, two different far pointers can point to the same address.

huge: a 32 bit pointer in which the segment is "normalised" so that no two far pointers point to the same address unless they have the same value.

#####

Normalize pointer- In 8086 programming (MS DOS), a far pointer is *normalized* if its offset part is between 0 and 15 (0xF).

#####

**Red-black trees-** are self-balancing, and so can insert, delete, and search in O(log n) time. Other types of balanced trees (e.g. AVL trees) are often slower for insert and delete operations.

In addition, the code for red-black trees tends to be simpler.

Red Black Tree is a special type of self balancing binary search tree. This is used as Syntax Trees in major compilers and as implementations of Sorted Dictionary.

Properties-

In addition to the requirements imposed on a binary search trees, with red–black trees:<sup>[4]</sup>

1. A node is either red or black.
2. The root is black. (This rule is sometimes omitted. Since the root can always be changed from red to black, but not necessarily vice-versa, this rule has little effect on analysis.)
3. All leaves (NIL) are black. (All leaves are same color as the root.)
4. Both children of every red node are black.
5. Every simple path from a given node to any of its descendant leaves contains the same number of black nodes.

#####

BITWISE OPERATOR-

Question- Write a program in C to know whether it is power of 2 or not.

```
int isPowerOfTwo (int x)
{
 return ((x != 0) && !(x & (x - 1)));
}
```

#####

Write a program in C to know whether a number is odd number or even using bit wise

Here is the simple program to find odd or even no using Bit wise operators:::::

The Logic is:: If the given number is ODD then it ends with 1 (LSB) in Binary format i.e num = 3, binary representation is 011.

If the given number is EVEN then it ends with 0 (LSB) in Binary format i.e num = 4, binary representation is 100.

Program is::::

```

main()
{
int num;

printf("Enter the Number\n");
scanf("%d",&num);

if(num&1)
printf("Entered Number is ODD\n");
else
printf("Entered Number is EVEN\n");
}
```

```
#####
```

**Question: You are given two numbers A and B. Write a program to count number of bits needed to be flipped to convert A to B.**

Solution:

1. Calculate XOR of A and B.  
 $a\_xor\_b = A \wedge B$
2. Count the set bits in the above calculated XOR result.  
 $countSetBits(a\_xor\_b)$

XOR of two number will have set bits only at those places where A differs from B.

Example:

A = 1001001  
B = 0010101  
 $a\_xor\_b = 1011100$   
No of bits need to flipped = set bit count in  $a\_xor\_b$  i.e. 4

Count set bits in an integer  
August 19, 2009

Write an efficient program to count number of 1s in binary representation of an integer.

1. Simple Method Loop through all bits in an integer, check if a bit is set and if it is then increment the set bit count. See below program.

```
/* Function to get no of set bits in binary
```

```
representation of passed binary no. */
```

```
int countSetBits(unsigned int n)
{
```

```
 unsigned int count = 0;
```

```
 while(n)
```

```
{
```

```
 count += n & 1;
```

```
 n >= 1;
```

```
}
```

```
return count;
```

```
}
```

```
#####
#####
```

### **Understanding “register” keyword in C**

February 20, 2010

Registers are faster than memory to access, so the variables which are most frequently used in a C program can be put in registers using register keyword. The keyword register hints to compiler that a given variable can be put in a register. It's compiler's choice to put it in a register or not. Generally, compilers themselves do optimizations and put the variables in register.

- 1) If you use & operator with a register variable then compiler may give an error or warning (depending upon the compiler you are using), because when we say a variable is a register, it may be stored in a register instead of memory and accessing address of a register is invalid. Try below program.

```
int main()
```

```
{
```

```
register int i = 10;
```

```
int *a = &i;
```

```
printf("%d", *a);
```

```
getchar();
```

```
return 0;
```

```
}
```

2) register keyword can be used with pointer variables. Obviously, a register can have address of a memory location. There would not be any problem with the below program.

```
int main()
```

```
{
```

```
int i = 10;
```

```
register int *a = &i;
```

```
printf("%d", *a);
```

```
getchar();
```

```
return 0;
```

```
}
```

3) Register is a storage class, and C doesn't allow multiple storage class specifiers for a variable. So, register can not be used with static . Try below program.

```
int main()
```

```
{
```

```
int i = 10;
```

```
register static int *a = &i;
```

```
printf("%d", *a);
```

```
getchar();
```

```
return 0;
```

```
}
```

4) There is no limit on number of register variables in a C program, but the point is compiler may put some variables in register and some not.

Please write comments if you find anything incorrect in the above article or you want to share more

information about register keyword.

```
#####
```

The first declaration:

```
const char * myPtr
```

declares a pointer to a constant character. You cannot use this pointer to change the value being pointed to:

```
char char_A = 'A';
const char * myPtr = &char_A;

*myPtr = 'J'; // error - can't change value of *myPtr
```

The second declaration,

```
char * const myPtr
```

declares a constant pointer to a character. The location stored in the pointer cannot change. You cannot change where this pointer points:

```
char char_A = 'A';
char char_B = 'B';
char * const myPtr = &char_A;

myPtr = &char_B; // error - can't change address of myPtr
```

The third declares a pointer to a character where both the pointer value and the value being pointed at will not change.

constant pointer to constant

```
const int *const ptr;
```

Above declaration is constant pointer to constant variable which means we cannot change value pointed by pointer as well as we cannot point the pointer to other variable

```
#####
```

### Returned values of printf() and scanf()

For example, below program prints geeksforgeeks 13

```

int main()
{
 printf(" %d", printf("%s", "geeksforgeeks"));
 getchar();
}

```

Irrespective of the string user enters, below program prints 1.

```

int main()
{
 char a[50];
 printf(" %d", scanf("%s", a));
 getchar();
}

```

```
#####
#####
```

### Que- Nth node from the end of a Linked List

Method 1 (Use length of linked list)

- 1) Calculate the length of Linked List. Let the length be len.
- 2) Print the  $(len - n + 1)$ th node from the begining of the Linked List.

Method 2 (Use two pointers)

Maintain two pointers – reference pointer and main pointer. Initialize both reference and main pointers to head. First move reference pointer to  $n$  nodes from head. Now move both pointers one by one until reference pointer reaches end. Now main pointer will point to nth node from the end. Return main pointer.

### Que- Function to check if a singly linked list is palindrome

### Que-Write a function to get the intersection point of two Linked Lists.

Method 1(Simply use two loops)

Use 2 nested for loops. Outer loop will be for each node of the 1st list and inner loop will be for 2nd list. In the inner loop, check if any of nodes of 2nd list is same as the current node of first linked list. Time complexity of this method will be  $O(mn)$  where  $m$  and  $n$  are the number of nodes in two lists.

### Method 2 (Mark Visited Nodes)

This solution requires modifications to basic linked list data structure. Have a visited flag with each node. Traverse the first linked list and keep marking visited nodes. Now traverse second linked list, If you see a visited node again then there is an intersection point, return the intersecting node. This solution works in  $O(m+n)$  but requires additional information with each node. A variation of this solution that doesn't require modification to basic data structure can be implemented using hash. Traverse the first linked list and store the addresses of visited nodes in a hash. Now traverse the second linked list and if you see an address that already exists in hash then return the intersecting node.

### Method 3(Make circle in first list)

Thanks to Saravanan Man for providing below solution.

1. Traverse the first linked list(count the elements) and make a circular linked list. (Remember last node so that we can break the circle later on).
2. Now view the problem as find the loop in the second linked list. So the problem is solved.
3. Since we already know the length of the loop(size of first linked list) we can traverse those many number of nodes in second list, and then start another pointer from the beginning of second list. we have to traverse until they are equal, and that is the required intersection point.
4. remove the circle from the linked list.

Method 4 (Traverse both lists and compare addresses of last nodes) This method is only to detect if there is an intersection point or not. (Thanks to NeoTheSaviour for suggesting this)

- 1) Traverse the list 1, store the last node address
- 2) Traverse the list 2, store the last node address.
- 3) If nodes stored in 1 and 2 are same then they are intersecting.

Time complexity of this method is  $O(m+n)$  and used Auxiliary space is  $O(1)$

#####

### Bitwise Operators:

[http://en.wikipedia.org/wiki/Bitwise\\_operation](http://en.wikipedia.org/wiki/Bitwise_operation)

What purpose do the bitwise AND, OR, XOR and the Shift operators serve?

Write a C program to count the bits set (bit value 1 ) in an integer? Find out and compare different possibilities?

Check if the 20th bit of a 32 bit integer is on or off?

Write a program in C to know whether it is power of 2 or not.

Write a program in C to know whether a number is odd number or even using bit wiseoperator.

Write a functionsetbits(x,p,n,y)that returns x with then bits that begin at position p set to therightmost n bits of y, leaving the other bits unchanged.

You are given two numbers A and B. Write a program to count number of bits needed to be flipped to convert A to B.

How to reverse the bits in an interger?

How to reverse the odd bits of an integer?

How would you count the number of bits set in a floating point number?

#####

## Storage for Strings in C

March 3, 2010

In C, a string can be referred either using a character pointer or as a character array.

Strings as character arrays

```
char str[4] = "GfG"; /*One extra for string terminator*/
/* OR */
char str[4] = {'G', 'f', 'G', '\0'}; /* '\0' is string terminator */
```

When strings are declared as character arrays, they are stored like other types of arrays in C. For example, if str[] is an auto variable then string is stored in stack segment, if it's a global or static variable then stored in data segment, etc.

Strings using character pointers

Using character pointer strings can be stored in two ways:

1) Read only string in a shared segment.

When string value is directly assigned to a pointer, in most of the compilers, it's stored in a read only block (generally in data segment) that is shared among functions.

```
char *str = "GfG";
```

In the above line "GfG" is stored in a shared read only location, but pointer str is stored in a read-write memory. You can change str to point something else but cannot change value at present str. So this kind of string should only be used when we don't want to modify string at a later stage in program.

2) Dynamically allocated in heap segment.

Strings are stored like other dynamically allocated things in C and can be shared among functions.

```
char *str;
int size = 4; /*one extra for '\0'*/
str = (char *)malloc(sizeof(char)*size);
*(str+0) = 'G';
*(str+1) = 'f';
*(str+2) = 'G';
*(str+3) = '\0';
```

Let us see some examples to better understand above ways to store strings.

### Example 1 (Try to modify string)

The below program may crash (gives segmentation fault error) because the line \*(str+1) = 'n' tries to write a read only memory.

```
int main()
{
 char *str;
 str = "GfG"; /* Stored in read only part of data segment */
 (str+1) = 'n'; / Problem: trying to modify read only memory */
 getchar();
 return 0;
}
```

Below program works perfectly fine as str[] is stored in writable stack segment.

```
int main()
{
 char str[] = "GfG"; /* Stored in stack segment like other auto variables */
 (str+1) = 'n'; / No problem: String is now GnG */
 getchar();
 return 0;
}
```

Below program also works perfectly fine as data at str is stored in writable heap segment.

```
int main()
{
 int size = 4;

 /* Stored in heap segment like other dynamically allocated things */
 char *str = (char *)malloc(sizeof(char)*size);
 *(str+0) = 'G';
 *(str+1) = 'f';
 *(str+2) = 'G';
 *(str+3) = '\0';
 (str+1) = 'n'; / No problem: String is now GnG */
 getchar();
 return 0;
}
```

```
}
```

Example 2 (Try to return string from a function)

The below program works perfectly fine as the string is stored in a shared segment and data stored remains there even after return of getString()

```
char *getString()
{
 char *str = "GfG"; /* Stored in read only part of shared segment */

 /* No problem: remains at address str after getString() returns*/
 return str;
}

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

The below program also works perfectly fine as the string is stored in heap segment and data stored in heap segment persists even after return of getString()

```
char *getString()
{
 int size = 4;
 char *str = (char *)malloc(sizeof(char)*size); /*Stored in heap segment*/
 *(str+0) = 'G';
 *(str+1) = 'f';
 *(str+2) = 'G';
 *(str+3) = '\0';

 /* No problem: string remains at str after getString() returns */
 return str;
}

int main()
{
 printf("%s", getString());
 getchar();
```

```

 return 0;
}

```

But, the below program may print some garbage data as string is stored in stack frame of function getString() and data may not be there after getString() returns.

```

char *getString()
{
 char str[] = "GfG"; /* Stored in stack segment */

 /* Problem: string may not be present after getString() returns */
 return str;
}

int main()
{
 printf("%s", getString());
 getchar();
 return 0;
}

```

```
#####
#####
```

### Evaluation order of operands

August 14, 2010

Consider the below C/C++ program.

```

#include<stdio.h>
int x = 0;

int f1()
{
 x = 5;
 return x;
}

int f2()
{
 x = 10;
 return x;
}

int main()
{
 int p = f1() + f2();

```

```

printf("%d ", x);
getchar();
return 0;
}

```

What would the output of the above program - '5' or '10'?

The output is undefined as the order of evaluation of f1() + f2() is not mandated by standard. The compiler is free to first call either f1() or f2(). Only when equal level precedence operators appear in an expression, the associativity comes into picture. For example, f1() + f2() + f3() will be considered as (f1() + f2()) + f3(). But among first pair, which function (the operand) evaluated first is not defined by the standard.

```
#####
#####
```

## Comma in C and C++

August 29, 2010

In C and C++, comma (,) can be used in two contexts:

### 1) Comma as an operator:

The comma operator (represented by the token ,) is a binary operator that evaluates its first operand and discards the result, it then evaluates the second operand and returns this value (and type). The comma operator has the lowest precedence of any C operator, and acts as a sequence point.

```

/* comma as an operator */
int i = (5, 10); /* 10 is assigned to i*/
int j = (f1(), f2()); /* f1() is called (evaluated) first followed by f2().
The returned value of f2() is assigned to j */

```

### 2) Comma as a separator:

Comma acts as a separator when used with function calls and definitions, function like macros, variable declarations, enum declarations, and similar constructs.

```

/* comma as a separator */
int a = 1, b = 2;
void fun(x, y);

```

The use of comma as a separator should not be confused with the use as an operator. For example, in below statement, f1() and f2() can be called in any order.

```

/* Comma acts as a separator here and doesn't enforce any sequence.
Therefore, either f1() or f2() can be called first */
void fun(f1(), f2());
See this for C vs C++ differences of using comma operator.

```

You can try below programs to check your understanding of comma in C.

```

// PROGRAM 1
#include<stdio.h>
int main()
{
 int x = 10;

```

```

int y = 15;

printf("%d", (x, y));
getchar();
return 0;
}
// PROGRAM 2: Thanks to Shekhu for suggesting this program
#include<stdio.h>
int main()
{
 int x = 10;
 int y = (x++, ++x);
 printf("%d", y);
 getchar();
 return 0;
}
// PROGRAM 3: Thanks to Venki for suggesting this program
int main()
{
 int x = 10, y;

 // The following is equivalent to y = x++
 y = (x++, printf("x = %d\n", x), ++x, printf("x = %d\n", x), x++);

 // Note that last expression is evaluated
 // but side effect is not updated to y
 printf("y = %d\n", y);
 printf("x = %d\n", x);

 return 0;
}
#####

```

## Operands for sizeof operator

February 9, 2010

In C, sizeof operator works on following kind of operands:

- 1) type-name: type-name must be specified in parentheses.

`sizeof (type-name)`

- 2) expression: expression can be specified with or without the parentheses.

`sizeof expression`

The expression is used only for getting the type of operand and not evaluated. For example, below code prints value of i as 5.

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

```
int main()
{
```

```

int i = 5;
int int_size = sizeof(i++);
printf("\n size of i = %d", int_size);
printf("\n Value of i = %d", i);

getchar();
return 0;
}

```

Output of the above program:  
size of i = depends on compiler  
value of i = 5

```
#####
#####
```

### **Result of comma operator as l-value in C and C++**

January 19, 2011

Using result of comma operator as l-value is not valid in C. But in C++, result of comma operator can be used as l-value if the right operand of the comma operator is l-value.

For example, if we compile the following program as a C++ program, then it works and prints b = 30. And if we compile the same program as C program, then it gives warning/error in compilation (Warning in Dev C++ and error in Code Blocks).

```

#include<stdio.h>

int main()
{
 int a = 10, b = 20;
 (a, b) = 30; // Since b is l-value, this statement is valid in C++, but not in C.
 printf("b = %d", b);
 getchar();
 return 0;
}

```

C++ Output:  
b = 30

```
#####
#####
```

### **What is evaluation order of function parameters in C?**

June 11, 2010

It is compiler dependent in C. It is never safe to depend on the order of evaluation of side effects. For example, a function call like below may very well behave differently from one compiler to another:

```
void func (int, int);
```

```

int i = 2;
func (i++, i++);

```

There is no guarantee (in either the C or the C++ standard language definitions) that the increments will be evaluated in any particular order. Either increment might happen first. func might get the arguments '2, 3', or it might get '3, 2', or even '2, 2'.

```
#####
```

### Order of operands for logical operators

January 3, 2011

The order of operands of logical operators &&, || are important in C/C++.

In mathematics, logical AND, OR, EXOR, etc... operations are commutative. The result will not change even if we swap RHS and LHS of the operator.

In C/C++ (may be in other languages as well) even though these operators are commutative, their order is critical. For example see the following code,

```
// Traverse every alternative node
while(pTemp && pTemp->Next)
{
 // Jump over to next node
 pTemp = pTemp->Next->Next;
}
```

The first part pTemp will be evaluated against NULL and followed by pTemp->Next. If pTemp->Next is placed first, the pointer pTemp will be dereferenced and there will be runtime error when pTemp is NULL.

It is mandatory to follow the order. Infact, it helps in generating efficient code. When the pointer pTemp is NULL, the second part will not be evaluated since the outcome of AND (&&) expression is guaranteed to be 0.

```
#####
```

### Static functions in C

May 5, 2010

In C, functions are global by default. The "static" keyword before a function name makes it static. For example, below function fun() is static.

```
static int fun(void)
{
 printf("I am a static function ");
}
```

Unlike global functions in C, access to static functions is restricted to the file where they are declared. Therefore, when we want to restrict access to functions, we make them static. Another reason for making functions static can be reuse of the same function name in other files.

For example, if we store following program in one file file1.c

```
/* Inside file1.c */
static void fun1(void)
{
 puts("fun1 called");
}
```

And store following program in another file file2.c

```
/* Inside file2.c */
int main(void)
{
```

```

fun1();
getchar();
return 0;
}

```

Now, if we compile the above code with command “gcc file2.c file1.c”, we get the error “undefined reference to `fun1`” . This is because fun1() is declared static in file1.c and cannot be used in file2.c.

```
#####
#####
```

### **exit(), abort() and assert()**

July 9, 2010

exit()

void exit ( int status );

exit() terminates the process normally.

status: Status value returned to the parent process. Generally, a status value of 0 or EXIT\_SUCCESS indicates success, and any other value or the constant EXIT\_FAILURE is used to indicate an error. exit() performs following operations.

- \* Flushes unwritten buffered data.
- \* Closes all open files.
- \* Removes temporary files.
- \* Returns an integer exit status to the operating system.

The C standard atexit() function can be used to customize exit() to perform additional actions at program termination.

Example use of exit.

```

/* exit example */
#include <stdio.h>
#include <stdlib.h>

int main ()
{
 FILE * pFile;
 pFile = fopen ("myfile.txt", "r");
 if (pFile == NULL)
 {
 printf ("Error opening file");
 exit (1);
 }
 else
 {
 /* file operations here */
 }
 return 0;
}

```

abort()

void abort ( void );

Unlike exit() function, abort() may not close files that are open. It may also not delete temporary files and may not flush stream buffer. Also, it does not call functions registered with atexit().

This function actually terminates the process by raising a SIGABRT signal, and your program can include a handler to intercept this signal (see this).

So programs like below might not write “Geeks for Geeks” to “myfile.txt”

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
 FILE *fp = fopen("C:\\myfile.txt", "w");

 if(fp == NULL)
 {
 printf("\n could not open file ");
 getchar();
 exit(1);
 }

 fprintf(fp, "%s", "Geeks for Geeks");

 /* */
 /* */
 /* Something went wrong so terminate here */
 abort();

 getchar();
 return 0;
}
```

If we want to make sure that data is written to files and/or buffers are flushed then we should either use exit() or include a signal handler for SIGABRT.

assert()

void assert( int expression );

If expression evaluates to 0 (false), then the expression, sourcecode filename, and line number are sent to the standard error, and then abort() function is called. If the identifier NDEBUG (“no debug”) is defined with #define NDEBUG then the macro assert does nothing.

Common error outputting is in the form:

Assertion failed: expression, file filename, line line-number

```
#include<assert.h>
```

```
void open_record(char *record_name)
{
```

```
assert(record_name != NULL);
/* Rest of code */
}

int main(void)
{
 open_record(NULL);
}
#####
#####
```

Dear All,

C Questions discussed on 22april 2013 are listed below. I don't know upto what extend these solutions are correct. Please let me know if they are wrong

#### [Q1 Traverse binary tree breadth-wise](#)

Ans:

```
void print(BinaryTree *root)
{
if (!root) return;
queue<BinaryTree*> currentLevel, nextLevel;
currentLevel.push(root);
while (!currentLevel.empty())
{
BinaryTree *currNode = currentLevel.front();
currentLevel.pop();
if (currNode)
{
printf(currNode->data) ;
nextLevel.push(currNode->left);
nextLevel.push(currNode->right);
}
if (currentLevel.empty())
{
printf("end");
}
```

```
swap(currentLevel, nextLevel);
}
}
}
```

find out least common ancestor in tree

```
void find(root, node1 , node2)
{
if(root==null || node1 ==null || node2==null)
return null;
if(node1== root)
return node1;
if(node2==root)
return node2;
if(node1 < root && node2 < root)
{
return find(root->left,node1,node2);
}
elseif(node1 > root && node2 > root)
{
return find(root->right,node1,node2);
}
return root;
}
```

Dear All,

These are the C Questions discussed on 19-04-2013, listed below.

**Q1.) Reverse a string using bitwise operator?**

Ans: using xor operator.

```
Int main(){
Char x[] = "Manohar";
Char *string = x;
int len = strlen(string);
for (int i=0; i<len/2; i++){
x[len-i-1] ^= x[i];
x[i] ^= x[len-i-1];
x[len-i-1] ^= x[i];
}
}
```

**Q2.) printf("%d", -1 <<4); what is the output ?**

Ans:1 is represented in binary form is 0000 0000 0000 0001

-1 means 1's compliment                    1111 1111 1111 1110

Output is FFF0

**Q3.) Use of bit fields in C?**

Ans: Both C and C++ allow integer members to be stored into memory spaces smaller than the compiler would ordinarily allow. Width in bit fields explicitly declared and A bit field declaration may not use either of the type qualifiers, const or volatile.

**The following restrictions apply to bit fields. You cannot:**

- &#61623 Define an array of bit fields
- &#61623 Take the address of a bit field
- &#61623 Have a pointer to a bit field
- &#61623 Have a reference to a bit field

The following example demonstrates padding. The structure kitchen contains eight members totaling 16 bytes.

```
struct on_off {
 unsigned light : 1;
 unsigned toaster : 1;
 int count; /* 4 bytes */
 unsigned ac : 4;
 unsigned : 4;
 unsigned clock : 1;
 unsigned : 0;
 unsigned flag : 1;
} kitchen;
```

| Member Name          | Storage Occupied                         |
|----------------------|------------------------------------------|
| light                | 1 bit                                    |
| toaster              | 1 bit                                    |
| (padding -- 30 bits) | To the next int boundary                 |
| count                | The size of an int (4 bytes)             |
| ac                   | 4 bits                                   |
| (unnamed field)      | 4 bits                                   |
| clock                | 1 bit                                    |
| (padding -- 23 bits) | To the next int boundary (unnamed field) |
| flag                 | 1 bit                                    |
| (padding -- 31 bits) | To the next int boundary                 |

**Q4.) what is the output of given code ?**

```
Int x = 5;
Printf("%d %d %d",x,x<<2,x>>2);
```

Ans: 5 20 1

**Q5.) char \*ptr=NULL;**

```
Char arr[10]={"abcd"};
Strcpy(ptr,arr);
Printf("%s",ptr);
```

**What is the output?**

Ans: segmentation fault. Since pointer is not pointing to any valid memory.

**Q6.) Int x[]={1,2,3,4,5};**

```
Int *ptr, **ptr2;
Ptr = x;
Ptr2 = &ptr;
```

**How do you update x[2]= 10 using ptr2 ?**

Ans: ptr2 + 2 and \*\*ptr = 10;

Or

\*\*(ptr2 + 2) = 10;

**Q7.) Convert the expression ((A + B) \* C – (D – E) ^ (F + G)) to postfix notation.**

Ans: AB + C \* DE - - FG + ^

**Q8.) Predict the output from the given code?**

```
Struct abc{
int a;
float b;
}d;
main(){
d.a = 1;
printf("%f" , d.b);
}
```

Ans: 0.0

**Q9.) Predict the which one is not valid statement?**

- 1.++d+++e++;
- 2.f\*=g+=h=5;
- 3.L-- >> M << -- N;
- 4.int a(int a), b = 0, c =a((c=b,++b));
- 5.i- >j< -k

Ans: 5

Thank you!

Best Regards,  
Manohar

**Q4:**

```
int p1 = sizeof("HELLO");
Int p2 = strlen("HELLO");
```

printf("\n p1 : %d p2: %d \n",p1,p2);

**Output :**

```
P1 = 6;
P2 = 5 ;
```

If "char \*str = "smartplay";int p1 = sizeof(str);" then p1 is 8. Pointer size 8 will be return.

Please check Q4 again.

Hi! All,

Below are the list of questions discussed on C-session (Apr 16 & 17).

| Q.No | Author | Question                                                                                                 | Output                                                                                                                                            | Discussion                                                                                                                                          |
|------|--------|----------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------|
|      |        | <pre>char *const ptr; char ch1,ch2; ptr = &amp;ch1; ptr = &amp;ch2;  What is value stored in ptr ?</pre> | <pre>Compilation Error: day1.c:6:1: error: assignment of read-only variable 'ptr' day1.c:7:1: error: assignment of read-only variable 'ptr'</pre> | <pre>char *const ptr; ==&gt; Address is constant , value can be changed.  Char const *ptr; ==&gt; Value is constant , address can be changed.</pre> |

|   |   |                                                                                                                                                                                  |                                                                                   |                                                                                                                                                 |
|---|---|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------|
| 2 |   | char arr[7] = "HELLO123";<br>printf("\n arr : %s",arr);<br>printf("\n arr2 %c \n",*arr);                                                                                         | arr : HELLO12<br>arr2 : H                                                         |                                                                                                                                                 |
| 3 |   | Explain situation to use double pointer                                                                                                                                          | int *p;<br>func(&p);<br><br>void func(int **q){<br>*q = malloc(sizeof(int));<br>} |                                                                                                                                                 |
| 4 |   | What the value of p1 & p2;<br>char *str = "smartplay";<br>int p1 = sizeof(str);<br>Int p2 = strlen(str);<br><br>printf("\n p1 : %d p2: %d \n",p1,p2);                            | P1 = 6;<br>P2 = 5 ;                                                               | Sizeof operator returns the size of string including null character<br><br>Strlen function returns length of a string excluding null character. |
| 5 | a | char *str1 = "smartplay";<br>char *str2[] = "smartplay";<br><br>printf("\n str1 : %u str2: %u \n",sizeof(str1),sizeof(str2));                                                    | Compilation error at *str2[].<br><br>error: invalid initializer                   | str2 pointer should be memory allocated.<br><br>Char *str2 = (char *) malloc(size(5));<br><br>Do dynamic allocation to avoid such errors.       |
| 6 | D | #define TRUE 0<br>#define FALSE -1<br>#define NULL 0<br><br>void main(){<br><br>if(TRUE) printf("TRUE");<br><br>else if(FALSE) printf("FALSE");<br><br>Else printf("NULL");<br>} | FALSE                                                                             | -1 = FF Consider as TRUE                                                                                                                        |

|   |                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |                                                                                                                 |                                                                                                                                                                                                                               |
|---|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 7 | <pre>enum actor {     abc = 5,     xyz = -2,     gcd,     hcf };  void main(){      enum actor a =0;     switch(a) {         case abc:             printf("abc");             break;         case xyz:             printf("xyz");             break;         case gcd:             printf("gcd");             break;         case hcf:             printf("hcf");             break;         default:             printf("default");             break;     } }</pre> | hcf                                                                                                             | xyz = -2,<br>gcd, // gets -1 (xyz + 1)<br>hcf // gets 0 ( gcd +1)                                                                                                                                                             |
| 8 | <pre>switch(*(1 + "AB" "CD" + 1)) {     case 'A':         printf("A");         break;     case 'B':         printf("B");         break;     case 'C':         printf("C");         break;     case 'D':         printf("D");         break;     default:         printf("default");         break; }</pre>                                                                                                                                                            | C                                                                                                               | 1 +ABCD + 1<br>2 + ABCD<br><br>A = 0 , B = 1 C = 2 , D = 3<br><br>Expected value is 2 , So it reach case C:<br><br>If switch((1 + "AB" "CD" + 1))<br>then its an compilation error :<br>error: switch quantity not an integer |
| 9 | <pre>struct student {     int rno;     char name[10]; }st;  void main(){      st.rno = 10;     <u>st.name</u> = "hello"; }</pre>                                                                                                                                                                                                                                                                                                                                      | Compilation Error :<br><br>error: incompatible types<br>when assigning to type<br>'char[10]' from type 'char *' | <u>st.name</u> = "hello"; is not allowed<br><br>strcpy( <u>st.name</u> ,"hello"); is allowed                                                                                                                                  |

|    |   |                                                                                             |                                                                                                                                              |                         |
|----|---|---------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------|-------------------------|
| 10 | G | <pre>int xx = NULL; int yy = 10; int zz = 0;  zz = xx + yy;  printf("Z : %d \n ",zz);</pre> | Z = 10;                                                                                                                                      | NULL Considered as a 0. |
| 11 |   | <pre>printf("char : %c ",59); printf("int : %d ",59); printf("string : %s ",59);</pre>      | Char - ; Int - 59 String - Segmentation fault                                                                                                |                         |
| 1  |   | <pre>int i = 4,5,6; Int I = (4,5,6);</pre>                                                  | First statement - Not allowed - error: expected identifier or '(' before numeric constant.<br><br>Second statement - I = 6 will be assigned. |                         |
| 13 |   | <pre>int I = -1; while(+(+i) != 0) { ++i; }</pre>                                           | I = 0;                                                                                                                                       |                         |

**U**

sorry for wrong answer in 1st question. please check it.

Q 1) if else statement is better than switch case if yes then why?

switch statement is better than if-else, because compiler is better in optimizing a switch-statement than an if-statement.

The compiler doesn't know if the order of evaluating the if-statements is important to you, and can't perform any optimizations there. You could be calling methods in the if-statements, influencing variables. With the switch-statement it knows that all clauses can be evaluated at the same time and can put them in whatever order is most efficient.

Q 1) if else statement is better than switch case if yes then why?

If else statement is better than switch, because compiler is better in optimizing a switch-statement than an if-statement.

The compiler doesn't know if the order of evaluating the if-statements is important to you, and can't perform any optimizations there. You could be calling methods in the if-statements, influencing variables. With the switch-statement it knows that all clauses can be evaluated at the same time and can put them in whatever order is most efficient.

Q 2)

```
main()
{
int i=5;
change();
int i=10;
```

```
printf("%d",i)
}
change()
{
?(need to write code here)
}
```

in above statement print the 5 with respect to the change function.

Ans. #define printf(a,b) print("5");

Description:

defined in above of main function is also It Prints 5.

Printf function is a Built in function Even it can be used as a User defined function.

Q 3)

```
#define printd(expr) printf(#expr " %d",expr)
main()
{
int a=10,b=2;
printd(a/b);
}
```

What is o/p?

Ans: a/b=5;  
#expr treats that variable names.

Q.1 write a function which takes 7 as input and returns 4 and vice versa ?

```
A.1 int func1(int p) // input p should be 7 or 4
{
int i;
i=(p^3);
return i;
}
```

Q.2 what is stack flow after the statement being executed ?

OR

How these four things gets into picture 1. Compiler 2. C Language 3. Glibc 4. Kernel after the below statement?

```
char *p=((char *)malloc (10));
```

A.2

1. C Language :- It specifies the header file <stdlib.h> ,which will check the prototype.

2. Compiler :- It takes the output of the preprocessor, and the source code, and generates assembler source

code it is kept in an object file.

3. Glibc :- The glibc package contains standard libraries. In order to save disk space and memory, as well as to make upgrading easier, common system code is kept in one place and shared between programs. This particular package contains the most important sets of shared libraries: the standard C library and the standard math library. Without these two libraries, a Linux system will not function

So object file generated now gets linked with the Glibc where in implementation of malloc is being defined.

There are other APIs like uclibc , bionic libc which can be linked with different OS (Linux, BCD, QNX) in .Config file while building kernel, depending on the requirement of the user.

3. Kernel :- Glibc calls the kernel via brk() system call.

1. Allocated 2D memory with malloc.

Ans: Pointers can be easily used to create a 2D array in C using malloc. The idea is to first create a one dimensional array of pointers, and then, for each array entry, create another one dimensional array. Here's a sample code:

```
double**theArray;
theArray=(double**)malloc(arraySizeX*sizeof(double*));
for(int i=0; i<arraySizeX;i++)
theArray[i]=(double*)malloc(arraySizeY*sizeof(double));
```

2.

```
int * fun()
{
static int i;
i=20;
return (&i);
}
```

Ans: its depend on OS implementation because static variable store in Data segment but it will try to return the value from stack so some times it works some time it does not work because it is dependent on how OS have property of memory distribution.

3.

```
int fun(int x)
{
int y;
y=x-1;
return((x&y)==0);
}
```

Ans: if value is 2 power n then it will return 1 else return 0 ;

sol: case 1: x=4 and then y=4-1=3

x=0100 and y =0011

So ((0100 & 0011)==0) it is true it will return 1.

Case2: x=6 and the y=5

x=0110 and y=0101  
So ( (0110 & 0101)==0) it is not true so it will return 0.  
conclusion: if value is 2 power n then it will return 1 else return 0 ;  
-----

Subject: RE: C Sessions: Day-4 Questions

C session (Tuesday,09.04.2013)

Que- How can we pass entire array (all element of array) to the function.

Ans- for passing an array, wrap it in a structure:

for example-

```
struct xyz
{
int array[100];
};

int somefunc(struct xyz large)
{
...
}
void anotherfunc(void)
{
struct xyz a;
printf("%d\n", somefunc(a));
}
#####
#####
```

Que- What is the O/P of following program?

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

```

main()
{
char arr[]="string";

printf("%c",*arr);

arr++;
printf("%c",*arr);

}

```

Ans- Error !!!  
error Line 9 : '++' needs l-value

A side effect of arr++ is an attempt to change the value of arr.we can't use statement like , 'arr++' because , arr is a constant pointer to the allocated memory

```
#####
#####
```

Q.1 predict the output for this code snippet with reason?

```

main()
{
int (*ptr)[5];

char *p;

int arr[5]={1,2,3,4,5};
ptr=arr;
p=arr;
printf("ptr=%u \n p=%u",ptr,p);
p++;
ptr++;
printf("ptr=%u \n p=%u",ptr,p);

printf("*p=%u\n *ptr=%u",*p,*ptr);

printf("**ptr=%u\n",**ptr);

}

```

A.1 ptr=2402524240 p=2402524240

ptr=2402524260 p=2402524241 ( ptr is pointer to array so increments by 20 bytes int size x array size , while p is pointer to char so increments by 1 byte)

\*p=0 ( p is char pointer so it takes first byte of 4 bytes assigned that is zero here

0000 0000 0000 0001 , value can be diff if it is big endian)  
\*ptr=2402524260 ( single dereference cant give u value as it is pointer to array,we have to double dereference it for getting value)

\*\*ptr= 0; ( it gives value to next array memory is not assigned so it can take any value)

#####

Que- X % Y (X modulus Y )

Given- y is a power of 2.

calculate X % Y without using % operator.

Solution-

x modulo y = (x & (y - 1))

#####

Que- int a= some bit stream;

int b = another bit stream;

only 1 bit difference between 'a' and 'b'. find out the position of mismatch bit.

Ans-

```
#include <iostream>
using namespace std;
int main()
{
int a=1123,b=1091; // declared a and b with integer and difference of their bits is only 1
int c,d;
int count=0;
c= a ^ b ; //
d= c-1;
if((c & d) ==0)
{
while(c !=0)
{
count++;
c=(c>>1);
}
}
cout<<count; // return position where bits are mismatched
}
```

#####

Que-

I have a bit stream and without using for loop, I want to set m bit to n bit.  
(m >n)  
eg:  
Bitstream = 10 10 10 10 01 00  
m= 10  
n=4,  
O/P- 00 01 11 11 00 00

Solution :  
 $(-1 \ll n) \& ((1 \ll m) - 1)$

Note: given solution set all the bit within the range but turn off the bits which are beyond the range. to overcome this problem we can use this logic.

num | ( ( (1<<( m- n+1 )-1 ) << ( n-1 ) )

#####
#

Subject: C Sessions: Day-4 Questions

strtok  
char \*strtok(char \*str, const char \*delim);  
The strtok() function parses a string into a sequence of tokens. On the first call to strtok() the string to be parsed should be specified in str. In each subsequent call that should parse the same string, str should be NULL. The delim argument specifies a set of bytes that delimit the tokens in the parsed string. The caller may specify different strings in delim in successive calls that parse the same string. Each call to strtok() returns a pointer to a null-terminated string containing the next token. This string does not include the delimiting byte. If no more tokens are found, strtok() returns NULL.

```
#include<string.h>

int main(){
char str[]="hello-this is rider";
char *pch;
char* a[10];
int i=0,j;
```

```

pch=strtok(str," ");
//we can add more special character

while(pch!=NULL){
printf("%s\n",pch);
a[i]=pch;
i++;
pch=strtok(NULL," ");
}

reverseWords(a,i);
printf("Reverse:\n");

for(j=0;j<i;j++)
printf("%s\n", a[j]);
return 0;
}

```

O/P:

```

rider
is
hello-this

```

```

void main()
{
int a[2][3][2] = {{{2,4},{7,8},{3,4}},{ {2,2},{2,3},{3,4}}};
printf("0x%u 0x%u 0x%u %d \n",a,*a,**a,***a);
printf("0x%u 0x%u 0x%u %d \n",a+1,*a+1,**a+1,***a+1);
}

```

O/P:

```

0x382857360 0x382857360 0x382857360 2
0x382857384 0x382857368 0x382857364 3

```

Explanation:

The base address of array is 0x382857360.

As we know element at location [1][1][1] can be accessed as “a[1][1][1]” or \*( \*( \*(a + 1) + 1) + 1).

T \*p; // p is a pointer to an object of type T

When a pointer p is pointing to an object of type T, the expression \*p is of type T.

For example a is of type array of 2 two dimensional arrays.

The type of the expression \*a is “array of arrays (i.e. two dimensional array)”.

Based on the above concept translating the expression \*( \*( \*(buffer + 1) + 1) + 1).

For a three-dimensional array, reference to an array has type "pointer to array of 3 ints, of which each element as an array of 2 ints".

Therefore, a+1 is pointing to the memory location of first element of the second 2D array, a.

So,

a - An array of 2 two dimensional arrays, i.e. its type is “array of 2 two dimensional arrays”.

a + 1 - displacement for 2nd element in the array of 5 two dimensional arrays.

i.e  $a+1$  is equivalent to  $a[1][0][0]$

Hence  $382857360 + 3 * (2 \text{ ints} * 4 \text{ bytes}) = 382857384$

Similarly, for  $*a+1$

$*(a + 0) + 1$  - displacement to access 1th element in the array of 3 one dimensional arrays.

Therefore  $*a+1$  is equivalent to  $a[0][1][0]$

Similarly, for  $**a+1$

$*(*(a + 0) + 0) + 1$  - displacement to get element at 1th position in the single dimension array of integers.

Therefore  $**a+1$  is equivalent to  $a[0][0][1]$

Similarly, for  $***a+1$  is equivalent to value at  $a[0][0][1]$

$*(*(*(a + 0) + 0) + 0)$  - accessing the element at 0th position (the overall expression type is int now).

### String Literals

```
char *string = "English"; /* string literal */
char array[] = "English"; /* Char array not a string literal */
array[0] = 'V' ; /*Legal, no error*/
string[0] = 'V'; /* Undefined */
```

ANSI C standard defines String literals as " A string has type ``array of characters" and storage class static and is initialized with the given characters. Whether identical string literals are distinct is implementation defined, and the behavior of a program that attempts to alter a string literal is undefined." Trying to modify string literal will give segmentation fault on Linux however it may work on other OS or on processors which do not have memory protection.

In gcc/Linux string literal [char \*ptr] is stored read-only area hence you get a segmentation fault. But if non gcc compiler stores string literal in Read-Write area OR if the processor does not have MMU and OS does not offer memory protection this program will work without any error. This is because C language standard does not clearly define what should be result of code trying to modify string literals; C standard had left it to compilers/OS/Processor to use in best way they can.

Note that

### Structure alignment

1. Memory is always byte addressable.
2. For speed optimization compiler by default uses alignment which is same as data bus size. For ex. In case of machine with 32-bit data bus alignment is done in multiples of 4-bytes.
3. In case of data types smaller than size of data bus padding bytes are added so that each variable is aligned.
4. Problem with padding: In real communication and in storage padding is never used as it is unnecessary wastage of time & space. But if the padded structures used to reference protocol header or in media files data it could lead to incorrect results because of difference in alignment of actual data verses padded structures.
5. To avoid such issues compilers provide #pragma pack [see compiler document for specific syntax]. With this pragma programmer can instruct compiler not to add any padding and all the structures must be packed even though it mean compromise of efficiency.

### Precedence & Associativity

```
#include<stdio.h>
main()
{
int a =10, b=20, c=0;
c=a+++b; /*No spaces in between*/
printf("a=%d, b=%d, c=%d\n",a,b,c);

return 0;
}
```

ANS:a=11, b=20 c=30

1. This is because ++ has higher precedence than +
  2. ++ has "Right to Left" associativity hence it gets associated with a and becomes a++ i.e. post increment.
- 
- 

### String Literals

```
char *string = "English"; /* string literal */
char array[] = "English"; /* Char array not a string literal */
array[0] = 'V' ; /*Legal, no error*/
string[0] = 'V'; /* Undefined */
```

ANSI C standard defines String literals as " A string has type ``array of characters" and storage class static and is initialized with the given characters. Whether identical string literals are distinct is implementation defined, and the behavior of a program that attempts to alter a string literal is undefined." Trying to modify string literal will give segmentation fault on Linux however it may work on other OS or on processors which do not have memory protection.

In gcc/Linux string literal [char \*ptr] is stored read-only area hence you get a segmentation fault. But if non gcc compiler stores string literal in Read-Write area OR if the processor does not have MMU and OS does not offer memory protection this program will work without any error. This is because C language standard does not clearly define what should be result of code trying to modify string literals; C standard had left it to compilers/OS/Processor to use in best way they can.

Note that

### Structure alignment

1. Memory is always byte addressable.
2. For speed optimization compiler by default uses alignment which is same as data bus size. For ex. In case of machine with 32-bit data bus alignment is done in multiples of 4-bytes.
3. In case of data types smaller than size of data bus padding bytes are added so that each variable is aligned.
4. Problem with padding: In real communication and in storage padding is never used as it is unnecessary wastage of time & space. But if the padded structures used to reference protocol header or in media files data it could lead to incorrect results because of difference in alignment of actual data verses padded structures.
5. To avoid such issues compilers provide #pragma pack [see compiler document for specific syntax]. With this pragma programmer can instruct compiler not to add any padding and all the structures must be packed even though it means compromise of efficiency.

Precedence & Associativity

```
#include<stdio.h>
main()
{
int a =10, b=20, c=0;
c=a+++b; /*No spaces in between*/
printf("a=%d, b=%d, c=%d\n",a,b,c);

return 0;
}
```

ANS:a=11, b=20 c=30

1. This is because ++ has higher precedence than +
2. ++ has "Right to Left" associativity hence it gets associated with a and becomes a++ i.e. post increment.

//

I2C protocol in details with timing diagrams

SPI protocol in details with timing diagrams

What is interrupt latency? How to reduce it?

What is difference between exceptions and interrupts?

Write C function to extract bits from position m to n where m > n and toggle those bits.

Write optimized function to convert little endian to big endian.

Write function to find if an integer is power of 2 or not

Write test cases for add function which adds three input values.

Different types of testing done during project work

Explain details of projects worked on so far. More questions on project.

Write and explain architecture of any latest project worked

Different modes in ARM processor? When is each mode entered?

What is system call? How are parameters passed in system call?

How are nested interrupts handled?

Explain complete boot process till linux is booted from power on reset.

Many questions on internals of boot process.

What are different boot devices available?

What are inputs for testing?

What test coverage tools have been used so far in projects worked?

1.What is volatile ?

2.What are Message queue & shared memory?

Which one is better explain with some example.

3.How to protect shared memory ?

4.Insert a node at a given position in a Linked list.

5.Endianness : How int i=10 stored in Little endian format

Which architecture uses Big endian.

int i,\*ptr;

ptr=&i;(Assuming &i is 1000)

ptr = ptr + 10;

What will be the value present in ptr.

ans : 1040

How to round off a number to a multiple of 1024 x 1024 (1Mb)(Using shifting operations)

6) int i;

Should be used in different files like a.c b.c c.c. How to achieve it.

7) Use of virtual memory

8) Which module of virtual memory does memory protection.

9) Delete a node from a double linked list pseudo code.

1. what is "little endian" and "big endian" and how to know this?

2. Is it possible to specify width of the data in strings? how? and what happens when that exceeded or reduced

3. what are the "unary operators?"

4. Is it possible to assaign a value to a variable before declaring?

5. can you write an inline code for finding largest of three numbers using ternary operator?

6. what is inline function in c?

7. write a program for '/' and '%' operators without using them.

8. Difference between return and goto functions

9. write a program to print pascal triangle

10. where main function return will go

11. where function declarations are present

12. who calls the main() function

13. Difference between recursion and iteration

14. Is it possible to multiply or division of two pointers?what are the arithmetic operations supported in pointers.

What is mutex ? why mutexes?

- 14.Why spinlocks are so important whats special with it?
- 15.Explain Priority Inversion with an example?
- 16.How do you handle events in Linux? Do You know about Signals and its handling?
- 17.What are ARM modes?
- 18.Why banked registers?
- 19.When Can u expect Stack Corruption?
- 20.How Can u decide the stack size?
- 21.Static and Global Variables ? Why ?
- 22.What is heap?
- 25.Write the code for Interrupt handling?
- 26.What are Scheduling Policies?
- 27.What are Cache Policies?Explain ?
- 28.What is the Size of Cache Line?

Write a C program to count number of 1's in a number?

What is semaphore , mutex and spinlock?

In ISR which synchronization mechanism is used and why?

Advantage of mutex over semaphore?

What is deadlock?

Can we detect deadlock?

What is priority inversion and how to avoid it?

In which cases deadlock situation occurs?

Write a program to find out sizeof a variable with out using sizeof operator?

What is malloc() and calloc()

Differences between malloc and calloc?

What is paging?

Form which area is dynamically allocated using malloc and whether that memory is virtual contiguous or virtual non contiguous or physical contiguous or physical non contiguous?

In kernel which function is used to allocate memory dynamically?

What is the use of wireshark?

What is USB?

What are the layers of USB in host side and client side?

32. fork()

fork()

fork

What is ISR?

7. Tell me what will happen when you are serving one interrupt and another interrupt comes.

8. Write a program for const variable? can you change the value of the const variable in program?  
write the program?

9.what is little endian and Big endian and write a program to in which 32 bit no changed into big endian?

10.memory leaks?

12. How to avoid memory leak?

13. What is malloc, write prototype, what it returns?

14. What will happen when you are accessing NULL pointer.

15. What is linked list?

16. Give a scenario in which loop detects in Linked list?
  17. How to avoid and detect loop in linked list?
  18. Linked list traversal?
  19. What is bottom halves?
  20. Give an example where we use Mutex and Spin locks?
  21. Tell the difference between Mutex and spin locks?  
Implement sizeof macro.it has to work if you give any variable.(ptr,int,var).
- 

- . What is the difference in features between kernel 2.2, 2.4 and 2.6 ?
2. What are Static and Shared libraries ?
3. What is dynamic linking ? What is static linking ?
4. What are the advantages of Dynamic linking or Shared libraries ?
5. Does gcc search for both static and shared libraries ? Which is searched initially by gcc compiler ?
6. What should be done for Shared library based linking in gcc ?
7. What should be done for static library based linking in gcc ?
8. What is object file and what are symbols ?
9. Can you tell the memory layout based on Data,BSS,HEAP and STACK ?
10. What are the ways in which linux kernel can be compiled ?
11. How will get the driver added into the kernel ? What are Kconfig files ?
12. What is a kernel module ?
13. What is the difference between insmod and modprobe ?
14. How will you list the modules ?
15. How do you get the list of currently available drivers ?
16. How will you Access userspace memory from kernel ? What are the various methods ?
17. What is the use of ioctl(inode,file,cmd,arg) API ?
18. What is the use of the poll(file, polltable) API ?
19. What is the use of file->private\_data in a device driver structure ?
20. What is a device number ?
21. What are the two types of devices drivers from VFS point of view ?
22. What are character devices ?
23. How does the character device driver adds and remove itself from the kernel ?  
What is the use of register\_chrdev and unregister\_chrdev ?
24. What is the role of interrupts in a device driver ? How are interrupts handled in device driver ?
25. How will you make interrupt handlers as fast as possible ?
26. What are the types of softirqs ?
27. Difference between Timer Softirq and Tasklet Softirq ?
28. What are tasklets ? How are they activated ? when and How are they initialized ?
29. What is task\_struct and how are task states maintained ?
30. What is rwlock and spinlock ? Briefly explain about both of them ?
31. When will you use rwlock instead of spinlock ?
32. Can spinlock/rwlock be used in Interrupt handler ?
33. Tell about the Memory Layout of a Process in Linux .
34. How will you trace the system calls made into the kernel of lInux ?
35. What is mmap ? MMAP & malloc ? MMAP & brk ? MMAP adv & dis-adv.
36. Tell the relation between Malloc and MMAP
37. Advantages of MMAP over Read ?
38. Tell the role of brk() in malloc / Tell the relation between heap and brk?
39. Example of using MMAP and MUNMAP in C ?

40. Tell about the method/steps in Linux Kernel Compilation.
41. What is Kmalloc and how does it differ from normal malloc ? or Why can't we use malloc in kernel code ?
42. What happens as soon as a packet arrives from the network in Linux ?
43. What is a stack frame, stack pointer & frame pointer ?
44. What is a profiler ? Which one have you used ?
45. How do you determine the direction of stack growth ?

Describe initial process sequence while the system boots up.

1) BIOS 2) Master Boot Record (MBR) 3) LILO or GRUB 4) Kernel 5) init 6) Run Levels.....

What is a shell? What are shell variables?

A shell us an interface to the user of any operating system.....

Explain how the inode maps to data block of a file.

There are 13 block addresses in inode. The file descriptions – type of file, access rights.....

Explain some system calls used for process management.

The following are the system calls: fork() - For creating child process .....

Explain how to get/set an environment variable from a program.

An environment variable can get using the function getenv().....

Describe how a parent and child process communicates each other.

The inter communication between a child process and a parent process can be done through normal communication.....

What is a Daemon?

Daemon is the short form for Disk and Execution Monitor.....

What is 'ps' command for?

The shortage for “process status” is ps. This command is used to display the currently running processes on Linux/Unix systems.....

### How the Swapper works?

Moving the information from fast access memory and slow access memory and vice versa is known as swapping.....

### What is the difference between Swapping and Paging?

Swapping performs the whole process to transfer to the disk, where as paging performs.....

### What is Expansion swap?

Expansion swap is a part of hard disk. This is reserved for the purpose of storing chunks of a program.....

### What is Fork swap?

For creation of child process, fork() system call is invoked. At the time of processing the fork() call by parent.....

### What are the requirements for a swapper to work?

The functionality of a swapper is on the scheduling priority which is highest. The swapper searches.....

### What is 'the principle of locality'?

The next most data item or instruction is the closest to the current data item or instruction.....

### What is page fault? Its types.

One of the critical parts of code in the Linux kernel. It has a major influence on memory subsystem's performance.....

### Difference between the fork() and vfork() system call.

fork: Both the parent and child share all of the page tables until any one of them does a write.....

### What is BSS(Block Started by Symbol)?

UNIX linkers produce uninitialized data segments.....

### What is Page-Stealer process? Explain the paging states for a page in memory.

The pages that are eligible for swapping are found by the Page-Stealer' .....

### Explain the phases of swapping a page from the memory.

The phases of swapping a page from the memory are:.....

### What is Demand Paging? Explain the conditions for a machine to support Demand Paging.

The process of mapping a large address space into a relatively small amount of physical memory is known as demand paging.....

Difference between Fault Handlers and the Interrupt handlers.

Fault handlers can sleep, where as interrupt handlers cannot.....

What is validity fault? In what way the validity fault handler concludes?

Validity fault is the result of non setting of valid bits in main memory at the time of referring a page by a process .....

What is protection fault?

Protection fault is a name of an error. This error occurs when accessing storage space is tried.....

Explain how the Kernel handles both the page stealer and the fault handler.

When the memory shortage occurs then the page stealer and fault handler thrashes.....

What is ex and vi? Explain their purposes.

ex – the line editor mode of 'vi' editor. It allows to.....

What is kill()? Explain its possible return values.

kill() is a system call which stops a process. The return values of kill() are:.....

Explain the steps that a shell follows while processing a command.

The sequence of executing commands by shell are as follows:.....

What is the difference between cmp and diff commands? Provide an example for each.

Byte by byte comparison performed for two files comparison and displays the first mismatch byte.....

What is the use of 'grep' command? Provide an example

Grep stands for regular expression. 'grep' command is used to find the patterns in a text file provided by the user.....

Difference between cat and more command.

The file contents are displayed by 'cat' command.....

What is 'du' command? What is its use?

The du (disk usage) command is used to report the size of directory trees.....

Explain the various prompts that are available in a UNIX system.

UNIX supports 4 prompts: PS1: default prompt.....

Describe how the kernel differentiates device files and ordinary files.

There are 2 device files. They are character device file and block device file.....

Explain how to switch to a super user status to gain privileges.

The command ?su? is used to get super user status.....

Linux - What are Pipes? Explain use of pipes

NEXT>>

*Linux - What are Pipes? Explain use of pipes - August 21, 2008 at 22:00 pm by Rajmeet Ghai*

What are Pipes? Explain use of pipes.

### Answer

A pipe is a chain of processes so that output of one process (stdout) is fed an input (stdin) to another. UNIX shell has a special syntax for creation of pipelines. The commands are written in sequence separated by |. Different filters are used for Pipes like AWK, GREP.

e.g. sort file | lpr ( sort the file and send it to printer)

Uses of Pipe

Several powerful functions can be in a single statement

Streams of processes can be redirected to user specified locations using >

*Linux - What are Pipes? Explain use of pipes - August 21, 2008 at 22:00 pm by Rajmeet Ghai*

What are Pipes? Explain use of pipes.

Pipe is a symbol used to provide output of one command as input to another command. The output of the command to the left of the pipe is sent as input to the command to the right of the pipe. The symbol is |.

For example:

```
$ cat apple.txt | wc
```

In the above example the output of apple.txt file will be sent as input for wc command which counts the no. of words in a file. The file for which the no. of words counts is the file apple.txt.

Pipes are useful to chain up several programs, so that multiple commands can execute at once without using a shell script

Linux - What is Linux and why is it so popular?

[NEXT>>](#)

*Linux - What is Linux and why is it so popular? - August 21, 2008 at 22:00 pm by Rajmeet Ghai*

What is Linux and why is it so popular?

### **Answer**

Linux is an operating system that uses UNIX like Operating system. However, unlike UNIX, Linux is an open source and free software. Linux was originally created by Linus Torvalds and commonly used in servers.

Popularity of Linux is because of the following reasons

It is free and open source. We can download Linux for free and customize it as per our needs.

It is very robust and adaptable.

Immense amount of libraries and utilities

*Linux - What is Linux and why is it so popular? - May 11, 2009 at 13:00 pm by Vidya Sagar*  
*What is Linux and why is it so popular?*

Linux is a multiuser, multitask GUI based open source operating system developed by Linus Torvalds. Linus Torvalds has invited the community to enhance the Linux kernel and thousands of system programmers worked on to enhance.

Prior to Linux, there was UNIX. The desktop work stations from various companies were based on UNIX. Later numerous companies entered and each one of them had their own UNIX version. As the proprietary authority is owned by each company and the lack of central authority weaken UNIX. As Linux is free and runs on any PC platform it gained the popularity very quickly. The following are few more reasons for its popularity:

- People who are familiar with UNIX can work on Linux with ease and comfort.
- People who want great control over network security and on operating system

How does the linux file system work?

### **Answer**

Linux file structure is a tree like structure. It starts from the root directory, represented by '/', and then expands into sub-directories. All the partitions are under the root directory. If a partition is mounted (The mount point defines the place of a particular data set in the file system) anywhere apart from a "device", the system is not aware of the existence of that partition or device. Directories that are only one level below the root directory are often preceded by a slash, to indicate their position.

Explain file system of linux. The root "/" filesystem, /usr filesystem, /var filesystem, /home filesystem, /proc filesystem.

### Answer

**Root "/" file system:** The kernel needs a root file system to mount at start up. The root file system is generally small and should not be changed often as it may interrupt in booting. The root directory usually does not have the critical files. Instead sub directories are created. E.g. /bin (commands needed during bootup), /etc (config files) , /lib(shared libraries).

**/usr filesystem :** this file system is generally large as it contains the executable files to be shared amongst different machines. Files are usually the ones installed while installing Linux. This makes it possible to update the system from a new version of the distribution, or even a completely new distribution, without having to install all programs again. Sub directories include /bin, /include, /lib, /local (for local executables)

**/var filesystem :** this file system is specific to local systems. It is called as var because the data keeps changing. The sub directories include /cache/man (A cache for man pages), /games (any variable data belong to games), /lib (files that change), /log (log from different programs), /tmp (for temporary files)

**/home filesystem:** - this file system differs from host to host. User specific configuration files for applications are stored in the user's home directory in a file. UNIX creates directories for all users directory. E.g /home/my\_name. Once the user is logged in ; he is placed in his home directory.

**/proc filesystem :** this file system does not exist on the hard disk. It is created by the kernel in its memory to provide information about the system. This information is usually about the processes. Contains a hierarchy of special files which represent the current state of the kernel .Few of the Directories include /1 (directory with information about process num 1, where 1 is the identification number), /cpuinfo (information about cpu), /devices (information about devices installed), /filesystem (file systems configured), /net (information about network protocols), /mem (memory usage)

What is Kernel? Explain the task it performs.

**Answer**

Kernel is used in UNIX like systems and is considered to be the heart of the operating system. It is responsible for communication between hardware and software components. It is primarily used for managing the systems resources as well.

Kernel Activities:

The Kernel task manager allows tasks to run concurrently.

Managing the computer resources: Kernel allows the other programs to run and use the resources. Resources include i/o devices, CPU, memory.

Kernel is responsible for Process management. It allows multiple processes to run simultaneously allowing user to multitask.

Kernel has an access to the systems memory and allows the processes to access the memory when required. Processes may also need to access the devices attached to the system. Kernel assists the processes in doing so.

For the processes to access and make use of these services, system calls are used.

*What is Kernel? Explain the task it performs.*

Kernel is the component that is responsible for managing the resources of a computer system.

The tasks are:

- Provides the abstraction level for resources such as memory, processors, and I/O devices.
- Performs inter process communication
- Responds to system calls
- Provides methods for synchronization and communication between processes.

Explain each system calls used for process management in linux.

**Answer**

System calls used for Process management:

Fork () :- Used to create a new process  
Exec() :- Execute a new program  
Wait():- wait until the process finishes execution  
Exit():- Exit from the process  
Getpid():- get the unique process id of the process  
Getppid():- get the parent process unique id  
Nice():- to bias the existing property of process

*Explain each system calls used for process management in linux.*

Process management uses certain system calls. They are explained below.

1. To create a new process – fork () is used.
2. To run a new program = exec () is used.
3. To make the process to wait = wait () is used.
4. To terminate the process – exit () is used.
5. To find the unique process id – getpid () is used.
6. To find the parent process id – getppid () is used.
7. To bias the currently running process property – nice () is used.

[What are the process states in Linux?](#)

### Answer

#### Process states in Linux:

**Running:** Process is either running or ready to run

**Interruptible:** a Blocked state of a process and waiting for an event or signal from another process

**Uninterruptible:** a blocked state. Process waits for a hardware condition and cannot handle any signal

**Stopped:** Process is stopped or halted and can be restarted by some other process

**Zombie:** process terminated, but information is still there in the process table.

[What are the process states in Linux?](#)

The following are the process states:

1. **Running:** This is a state where a process is either in running or ready to run.
2. **Interruptible:** This state is a blocked state of a process which awaits for an event or a signal from another process
3. **Uninterruptible:** It is also a blocked state. The process is forced to halt for certain condition that a hardware status is waited and a signal could not be handled.
4. **Stopped:** Once the process is completed, this state occurs. This process can be restarted
5. **Zombie:** In this state, the process will be terminated and the information will still be available in the process table.

[What is Linux Shell? What is Shell Script?](#)

### Answer

Linux shell is a user interface used for executing the commands. Shell is a program the user uses for executing the commands. In UNIX, any program can be the users shell. Shell categories in Linux are:

Bourne shell compatible, C shell compatible, nontraditional, and historical.

A shell script, as the name suggests, is a script written for the shell. Script here means a programming language used to control the application. The shell script allows different commands entered in the shell to be executed. Shell script is easy to debug, quicker as compared to writing big programs. However the execution speed is slow because it launches a new process for every shell command executed. Examples of commands are cp, cn, cd.

[Linux - What is Linux Shell? What is Shell Script? - August 21, 2008 at 22:00 pm by Rajmeet Ghai](#)

[What is Linux Shell? What is Shell Script?](#)

Linux shell is the user interface to communicate with Linux operating system. Shell interprets the user requests, executes them. Shell may use kernel to execute certain programs. Shell Script: A shell script is a program file in which certain Linux commands are placed to execute one after another. A shell script is a flat text file. Shell scripts are useful to accept inputs and provide output to the user. Everyday automation process can be simplified by a shell script.

1. Allocated 2D memory with malloc.

Ans: Pointers can be easily used to create a 2D array in C using malloc. The idea is to first create a one dimensional array of pointers, and then, for each array entry, create another one dimensional array. Here's a sample code:

```
double**theArray;
theArray=(double**)malloc(arraySizeX*sizeof(double*));
for(int i=0; i<arraySizeX;i++)
theArray[i]=(double*)malloc(arraySizeY*sizeof(double));
```

-----  
2.

```
int * fun()
{
 static int i;
 i=20;
 return (&i);
}
```

Ans: its depend on OS implementation because static variable store in Data segment but it will try to return the value from stack so some times it works some time it does not work because it is dependent on how OS have property of memory distribution.

-----  
3.  
int fun(int x)
{
int y;
y=x-1;
return((x&y)==0);
}

Ans: if value is 2 power n then it will return 1 else return 0 ;

sol: case 1: x=4 and then y=4-1=3  
x=0100 and y =0011  
So ((0100 & 0011)==0) it is true it will return 1.

Case2: x=6 and the y=5  
x=0110 and y=0101  
So ( (0110 & 0101)==0) it is not true so it will return 0.  
conclusion: if value is 2 power n then it will return 1 else return 0 ;

Que- How can we pass entire array (all element of array) to the function.

Ans- for passing an array, wrap it in a structure:

for example-

```
struct xyz
{
int array[100];
};

int somefunc(struct xyz large)
{
...
}
void anotherfunc(void)
{
struct xyz a;
printf("%d\n", somefunc(a));
}
#####
#####
```

Que- What is the O/P of following program?

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

```
main()
{
char arr[]="string";
printf("%c",*arr);

arr++;
printf("%c",*arr);

}
```

Ans- Error !!!  
error Line 9 : '++' needs l-value

A side effect of arr++ is an attempt to change the value of arr.we can't use statement like , 'arr++'  
because , arr is a constant pointer to the allocated memory

```
#####
#####
```

Q.1 predict the output for this code snippet with reason?

```
main()
{
int (*ptr)[5];

char *p;

int arr[5]={1,2,3,4,5};
ptr=arr;
p=arr;
printf("ptr=%u \n p=%u",ptr,p);
p++;
ptr++;
printf("ptr=%u \n p=%u",ptr,p);

printf("*p=%u\n *ptr=%u",*p,*ptr);

printf("**ptr=%u\n",**ptr);
}

A.1 ptr=2402524240 p=2402524240
```

ptr=2402524260 p=2402524241 ( ptr is pointer to array so increments by 20 bytes int size x array  
size , while p is pointer to char so increments by 1 byte)

\*p=0 ( p is char pointer so it takes first byte of 4 bytes assigned that is zero here  
0000 0000 0000 0001 , value can be diff if it is big endian)  
\*ptr=2402524260 ( single dereference cant give u value as it is pointer to array,we have to double  
dereference it for getting value)

\*\*ptr= 0; ( it gives value to next array memory is not assigned so it can take any value)

```
#####
#####
```

Que-     X % Y   (X modulus Y )  
 Given- y is a power of 2.

calculate X % Y without using % operator.

Solution-

```
x modulo y = (x & (y - 1))
```

```
#####
#####
```

Que- int a= some bit stream;  
 int b = another bit stream;  
 only 1 bit difference between 'a' and 'b'. find out the position of mismatch bit.  
 Ans-

```
#include <iostream>
using namespace std;
int main()
{
int a=1123,b=1091; // declared a and b with integer and difference of their bits is only 1
int c,d;
int count=0;
c= a ^ b ; //
d= c-1;
if((c & d) ==0)
{
while(c !=0)
{
count++;
c=(c>>1);
}
cout<<count; // return position where bits are mismatched
}
```

```
#####
#####
```

Que-  
 I have a bit stream and without using for loop, I want to set m bit to n bit.  
 (m >n)  
 eg:  
 Bitstream = 10 10 10 10 01 00  
 m= 10  
 n=4,  
 O/P-               00 01 11 11 00 00

Solution :  
 $(-1 \ll n) \& ((1 \ll m) - 1)$

Note: given solution set all the bit within the range but turn off the bits which are beyond the range.  
to overcome this problem we can use this logic.

```
num | ((1<<(m- n+1))-1) << (n-1)
```

```
#####
#####
```

```
strtok
char *strtok(char *str, const char *delim);
The strtok() function parses a string into a sequence of tokens. On the first call to strtok() the string to be parsed should be specified in str. In each subsequent call that should parse the same string, str should be NULL. The delim argument specifies a set of bytes that delimit the tokens in the parsed string. The caller may specify different strings in delim in successive calls that parse the same string. Each call to strtok() returns a pointer to a null-terminated string containing the next token. This string does not include the delimiting byte. If no more tokens are found, strtok() returns NULL.
```

```
#include<string.h>

int main(){
char str[]="hello-this is rider";
char *pch;
char* a[10];
int i=0,j;
pch=strtok(str," ");
//we can add more special character
```

```
while(pch!=NULL){
printf("%s\n",pch);
a[i]=pch;
i++;
pch=strtok(NULL," ");
}

reverseWords(a,i);
printf("Reverse:\n");

for(j=0;j<i;j++)
printf("%s\n", a[j]);
return 0;
}
```

```
O/P:
rider
is
hello-this
```

---



---

```
void main()
{
int a[2][3][2] = {{{2,4},{7,8},{3,4}},{ {2,2},{2,3},{3,4}}};
printf("0x%u 0x%u 0x%u %d \n",a,*a,**a,***a);
printf("0x%u 0x%u 0x%u %d \n",a+1,*a+1,**a+1,***a+1);
}
```

```
O/P:
```

```
0x382857360 0x382857360 0x382857360 2
0x382857384 0x382857368 0x382857364 3
```

Explanation:

The base address of array is 0x382857360.

As we know element at location [1][1][1] can be accessed as "a[1][1][1]" or \*( \*( \*(a + 1) + 1) + 1).

T \*p; // p is a pointer to an object of type T

When a pointer p is pointing to an object of type T, the expression \*p is of type T.

For example a is of type array of 2 two dimensional arrays.

The type of the expression \*a is "array of arrays (i.e. two dimensional array)".

Based on the above concept translating the expression \*( \*( \*(buffer + 1) + 1) + 1).

For a three-dimensional array, reference to an array has type "pointer to array of 3 ints, of which each element as an array of 2 ints".

Therefore, a+1 is pointing to the memory location of first element of the second 2D array, a.

So,

a - An array of 2 two dimensional arrays, i.e. its type is "array of 2 two dimensional arrays".

a + 1 - displacement for 2nd element in the array of 5 two dimensional arrays.

i.e a+1 is equivalent to a[1][0][0]

Hence 382857360 + 3\*(2 ints \* 4 bytes) = 382857384

Similarly, for \*a+1

\*( \*(a + 0) + 1 - displacement to access 1th element in the array of 3 one dimensional arrays.

Therefore \*a+1 is equivalent to a[0][1][0]

Similarly, for \*\*a+1

\*( \*( \*(a + 0) + 0) + 1 - displacement to get element at 1th position in the single dimension array of integers.

Therefore \*\*a+1 is equivalent to a[0][0][1]

Similarly, for \*\*\*a+1 is equivalent to value at a[0][0][1]

\*( \*( \*(a + 0) + 0) + 0) - accessing the element at 0th position (the overall expression type is int now).

### String Literals

```
char *string = "English"; /* string literal */
char array[] = "English"; /* Char array not a string literal */
array[0] = 'V' ; /*Legal, no error*/
string[0] = 'V'; /* Undefined */
```

ANSI C standard defines String literals as " A string has type ``array of characters'' and storage class static and is initialized with the given characters. Whether identical string literals are distinct is implementation defined, and the behavior of a program that attempts to alter a string literal is undefined." Trying to modify string literal will give segmentation fault on Linux however it may work on other OS or on processors which do not have memory protection.

In gcc/Linux string literal [char \*ptr] is stored read-only area hence you get a segmentation fault. But if non gcc compiler stores string literal in Read-Write area OR if the processor does not have MMU and OS does not offer memory protection this program will work without any error. This is because C language standard does not clearly define what should be result of code trying to modify string literals; C standard had left it to compilers/OS/Processor to use in best way they can.

Note that

### Structure alignment

1. Memory is always byte addressable.

2. For speed optimization compiler by default uses alignment which is same as data bus size. For ex. In case of machine with 32-bit data bus alignment is done in multiples of 4-bytes.
3. In case of data types smaller than size of data bus padding bytes are added so that each variable is aligned.
4. Problem with padding: In real communication and in storage padding is never used as it is unnecessary wastage of time & space. But if the padded structures used to reference protocol header or in media files data it could lead to incorrect results because of difference in alignment of actual data verses padded structures.
5. To avoid such issues compilers provide #pragma pack [see compiler document for specific syntax]. With this pragma programmer can instruct compiler not to add any padding and all the structures must be packed even though it mean compromise of efficiency.

```
Precedence & Associativity
#include<stdio.h>
main()
{
int a =10, b=20, c=0;
c=a++ + b; /*No spaces in between*/
printf("a=%d, b=%d, c=%d\n",a,b,c);

return 0;
}
```

ANS:a=11, b=20 c=30

1. This is because ++ has higher precedence than +
2. ++ has "Right to Left" associativity hence it gets associated with a and becomes a++ i.e. post increment.

---

Q 1) if else statement is better than switch case if yes then why?

If else statement is better than switch, because compiler is better in optimizing a switch-statement than an if-statement.

The compiler doesn't know if the order of evaluating the if-statements is important to you, and can't perform any optimizations there. You could be calling methods in the if-statements, influencing variables. With the switch-statement it knows that all clauses can be evaluated at the same time and can put them in whatever order is most efficient.

Q 2)

```
main()
{
int i=5;
change();
int i=10;
printf("%d",i)
}
change()
{
?(need to write code here)
}
```

in above statement print the 5 with respect to the change function.

Ans. #define printf(a,b) print("5");

Description:

defined in above of main function is also It Prints 5.

Printf function is a Built in function Even it can be used as a User defined function.

Q 3)

```
#define printd(expr) printf(#expr " %d",expr)
main()
{
int a=10,b=2;
printd(a/b);
}
```

What is o/p?

Ans: a/b=5;  
#expr treats that variable names.

Q.1 write a function which takes 7 as input and returns 4 and vice versa ?

```
A.1 int func1(int p) // input p should be 7 or 4
{
int i;
i=(p^3);
return i;
}
```

Q.2 what is stack flow after the statement being executed ?

OR

How these four things gets into picture 1. Compiler 2. C Language 3. Glibc 4. Kernel after the below statement?

```
char *p=((char *)malloc (10));
```

A.2

1. C Language :- It specifies the header file <stdlib.h> ,which will check the prototype.
2. Compiler :- It takes the output of the preprocessor, and the source code, and generates assembler source code it is kept in an object file.

3. Glibc :- The glibc package contains standard libraries. In order to save disk space and memory, as well as to make upgrading easier, common system code is kept in one place and shared between programs.

This particular package contains the most important sets of shared libraries: the standard C library and the standard math library. Without these two libraries, a Linux system will not function

So object file generated now gets linked with the Glibc where in implementation of malloc is being defined.

There are other APIs like uclibc, bionic libc which can be linked with different OS (Linux, BCD, QNX) in .Config file while building kernel, depending on the requirement of the user.

3. Kernel :- Glibc calls the kernel via brk() system call.

1. Allocated 2D memory with malloc.

Ans: Pointers can be easily used to create a 2D array in C using malloc. The idea is to first create a one dimensional array of pointers, and then, for each array entry, create another one dimensional array. Here's a sample code:

```
double**theArray;
theArray=(double**)malloc(arraySizeX*sizeof(double*));
for(int i=0; i<arraySizeX;i++)
theArray[i]=(double*)malloc(arraySizeY*sizeof(double));

```

2.

```
int * fun()
{
static int i;
i=20;
return (&i);
}
```

Ans: its depend on OS implementation because static variable store in Data segment but it will try to return the value from stack so some times it works some time it does not work because it is dependent on how OS have property of memory distribution.

```
3.
int fun(int x)
{
int y;
y=x-1;
return((x&y)==0);
}
```

Ans: if value is 2 power n then it will return 1 else return 0 ;

sol: case 1: x=4 and then y=4-1=3

x=0100 and y =0011

So ((0100 & 0011)==0) it is true it will return 1.

Case2: x=6 and the y=5

x=0110 and y=0101

So ((0110 & 0101)==0) it is not true so it will return 0.

Conclusion: if value is 2 power n then it will return 1 else return 0 ;

Ques- How can we pass entire array (all elements of array) to the function.

Ans- for passing an array, wrap it in a structure:

for example-

```
struct xyz
{
int array[100];
};

int somefunc(struct xyz large)
{
...
}
void anotherfunc(void)
{
struct xyz a;
printf("%d\n", somefunc(a));
}
#####
#####
```

Que- What is the O/P of following program?

```
#include<stdio.h>

main()
{
char arr[]="string";

printf("%c",*arr);

arr++;

printf("%c",*arr);

}
```

Ans- Error !!!  
error Line 9 : '++' needs l-value

A side effect of arr++ is an attempt to change the value of arr.we can't use statement like , 'arr++'  
because , arr is a constant pointer to the allocated memory

```
#####
#####
```

Q.1 predict the output for this code snippet with reason?

```
main()
{
int (*ptr)[5];

char *p;

int arr[5]={1,2,3,4,5};
ptr=arr;
p=arr;
printf("ptr=%u \n p=%u",ptr,p);
p++;
ptr++;
printf("ptr=%u \n p=%u",ptr,p);
```

```

printf("*p=%u\n *ptr=%u",*p,*ptr);

printf("**ptr=%u\n",**ptr);

}

A.1 ptr=2402524240 p=2402524240

ptr=2402524260 p=2402524241 (ptr is pointer to array so increments by 20 bytes int size x array
size , while p is pointer to char so increments by 1 byte)

*p=0 (p is char pointer so it takes first byte of 4 bytes assigned that is zero here
0000 0000 0000 0001 , value can be diff if it is big endian)
*ptr=2402524260 (single dereference cant give u value as it is pointer to array,we have to double
dereference it for getting value)

**ptr= 0; (it gives value to next array memory is not assigned so it can take any value)

#####

```

Que-     X % Y   (X modulus Y )  
 Given- y is a power of 2.

calculate X % Y without using % operator.

Solution-

```
x modulo y = (x & (y - 1))
```

```
#####
Que- int a= some bit stream;
int b = another bit stream;
only 1 bit diffrence between 'a' and 'b'. find out the position of mismatch bit.
Ans-
```

```
#include <iostream>
using namespace std;
int main()
{
int a=1123,b=1091; // declared a and b with integer and diffrence of their bits is only 1
int c,d;
int count=0;
c= a ^ b ; //
d= c-1;
if((c & d) ==0)
{
while(c !=0)
{
count++;
c=(c>>1);
}
}
cout<<count; // return position where bits are mismatched
}
```

```
#####
Que-
I have a bit stream and without using for loop, I want to set m bit to n bit.
(m >n)
eg:
Bitstream = 10 10 10 10 01 00
m= 10
n=4,
O/P-
 00 01 11 11 00 00
```

Solution :  
 $(-1 \ll n) \& ((1 \ll m) - 1)$

Note: given solution set all the bit within the range but turn off the bits which are beyond the range.  
 to overcome this problem we can use this logic.

```
num | (((1<<(m- n+1))-1) << (n-1))
```

```
#####
strtok
char *strtok(char *str, const char *delim);
The strtok() function parses a string into a sequence of tokens. On the first call to strtok() the string to be parsed should be specified in str. In each subsequent call that should parse the same string, str should be NULL. The delim argument specifies a set of bytes that delimit the tokens in the parsed string. The caller may specify different strings in delim in successive calls that parse the same string. Each call to strtok() returns a pointer to a null-terminated string containing the next token. This string does not include the delimiting byte. If no more tokens are found, strtok() returns NULL.
```

```
#include<string.h>

int main(){
char str[]="hello-this is rider";
char *pch;
char* a[10];
int i=0,j;
pch=strtok(str," ");
//we can add more special character

while(pch!=NULL){
printf("%s\n",pch);
a[i]=pch;
i++;
}
```

```

i++;
pch=strtok(NULL, " ");
}

reverseWords(a,i);
printf("Reverse:\n");

for(j=0;j<i;j++)
printf("%s\n", a[j]);
return 0;
}

```

O/P:  
rider  
is  
hello-this

---



---

```

void main()
{
int a[2][3][2] = {{{2,4},{7,8},{3,4}},{ {2,2},{2,3},{3,4}}};
printf("0x%u 0x%u 0x%u %d \n",a,*a,**a,***a);
printf("0x%u 0x%u 0x%u %d \n",a+1,*a+1,**a+1,***a+1);
}

```

O/B:  
0x382857360 0x382857360 0x382857360 2  
0x382857384 0x382857368 0x382857364 3

Explanation:

The base address of array is 0x382857360.

As we know element at location [1][1][1] can be accessed as "a[1][1][1]" or \*( \*( \*(a + 1) + 1) + 1).

T \*p; // p is a pointer to an object of type T  
When a pointer p is pointing to an object of type T, the expression \*p is of type T.  
For example a is of type array of 2 two dimensional arrays.  
The type of the expression \*a is "array of arrays (i.e. two dimensional array)".

Based on the above concept translating the expression \*( \*( \*(buffer + 1) + 1) + 1).

For a three-dimensional array, reference to an array has type "pointer to array of 3 ints, of which each element as an array of 2 ints".  
Therefore, a+1 is pointing to the memory location of first element of the second 2D array, a.

So,  
a - An array of 2 two dimensional arrays, i.e. its type is "array of 2 two dimensional arrays".  
a + 1 - displacement for 2nd element in the array of 5 two dimensional arrays.

i.e a+1 is equivalent to a[1][0][0]  
Hence 382857360 + 3\*(2 ints \* 4 bytes) = 382857384

Similarly, for \*a+1  
\*(a + 0) + 1 - displacement to access 1th element in the array of 3 one dimensional arrays.  
Therefore \*a+1 is equivalent to a[0][1][0]

Similarly, for \*\*a+1  
\*( \*(a + 0) + 0) + 1 - displacement to get element at 1th position in the single dimension array of integers.  
Therefore \*\*a+1 is equivalent to a[0][0][1]

Similarly, for \*\*\*a+1 is equivalent to value at a[0][0][1]  
\*( \*( \*(a + 0) + 0) + 0) - accessing the element at 0th position (the overall expression type is int

now) .

```
String Literals
char *string = "English"; /* string literal */
char array[] = "English"; /* Char array not a string literal */
array[0] = 'V' ; /*Legal, no error*/
string[0] = 'V'; /* Undefined */
```

ANSI C standard defines String literals as " A string has type ``array of characters'' and storage class static and is initialized with the given characters. Whether identical string literals are distinct is implementation defined, and the behavior of a program that attempts to alter a string literal is undefined." Trying to modify string literal will give segmentation fault on Linux however it may work on other OS or on processors which do not have memory protection.

In gcc/Linux string literal [char \*ptr] is stored read-only area hence you get a segmentation fault. But if non gcc compiler stores string literal in Read-Write area OR if the processor does not have MMU and OS does not offer memory protection this program will work without any error. This is because C language standard does not clearly define what should be result of code trying to modify string literals; C standard had left it to compilers/OS/Processor to use in best way they can.

Note that

Structure alignment

1. Memory is always byte addressable.
2. For speed optimization compiler by default uses alignment which is same as data bus size. For ex. In case of machine with 32-bit data bus alignment is done in multiples of 4-bytes.
3. In case of data types smaller than size of data bus padding bytes are added so that each variable is aligned.
4. Problem with padding: In real communication and in storage padding is never used as it is unnecessary wastage of time & space. But if the padded structures used to reference protocol header or in media files data it could lead to incorrect results because of difference in alignment of actual data verses padded structures.
5. To avoid such issues compilers provide #pragma pack [see compiler document for specific syntax]. With this pragma programmer can instruct compiler not to add any padding and all the structures must be packed even though it mean compromise of efficiency.

Precedence & Associativity

```
#include<stdio.h>
main()
{
int a =10, b=20, c=0;
c=a+++b; /*No spaces in between*/
printf("a=%d, b=%d, c=%d\n",a,b,c);

return 0;
}
```

ANS:a=11, b=20 c=30

1. This is because ++ has higher precedence than +
2. ++ has "Right to Left" associativity hence it gets associated with a and becomes a++ i.e. post increment.

Q.1 write a function which takes 7 as input and returns 4 and vice versa ?

A.1 int func1( int p) // input p should be 7 or 4

```
{
int i;
i=(p^3);
return i;
}
```

Q.2 what is stack flow after the statement being executed ?

OR

How these four things gets into picture 1. Compiler 2. C Language 3. Glibc 4. Kernel after the below statement?

```
char *p=((char *)malloc (10));
```

A.2

1. **C Language** :- It specifies the header file `<stdlib.h>` ,which will check the prototype.
2. **Compiler** :- It takes the output of the preprocessor, and the source code, and generates assembler source code it is kept in an object file.
3. **Glibc** :- The glibc package contains standard libraries.In order to save disk space and memory, as well as to make upgrading easier, common system code is kept in one place and shared between programs.  
This particular package contains the most important sets of shared libraries: the standard C library and the standard math library. Without these two libraries, a Linux system will not function

So object file generated now gets linked with the Glibc where in implementation of malloc is being defined.

There are other APIs like **uclibc** , **bionic libc** which can be linked with different **OS** (Linux, BCD, QNX) in .Config file while building kernel, depending on the requirement of the user.

3. **Kernel :-** Glibc calls the kernel via brk() system call.

1. Allocated 2D memory with malloc.

Ans: Pointers can be easily used to create a 2D array in C using malloc. The idea is to first create a one dimensional array of pointers, and then, for each array entry, create another one dimensional array. Here's a sample code:

```
double**theArray;
theArray=(double**)malloc(arraySizeX*sizeof(double*));
for(int i=0; i<arraySizeX; i++)
theArray[i]=(double*)malloc(arraySizeY*sizeof(double));
```

---

2.

```
int * fun()
{
static int i;
i=20;
return (&i);
}
```

Ans: its depend on OS implementation because static variable store in Data segment but it will try to return the value from stack so some times it works some time it does not work because it is dependent on how OS have property of memory distribution.

---

3.

```
int fun(int x)
{
int y;
y=x-1;
return((x&y)==0);
}
```

Ans: if value is 2 power n then it will return 1 else return 0 ;

sol: case 1: x=4 and then y=4-1=3

x=0100 and y =0011

So  $((0100 \& 0011)==0)$  it is true it will return 1.

Case2: x=6 and the y=5

x=0110 and y=0101

So  $((0110 \& 0101)==0)$  it is not true so it will return 0.

Conclusion: if value is  $2^n$  then it will return 1 else return 0 ;

---

Que- How can we pass entire array (all elements of array) to the function.

Ans- for passing an array, wrap it in a structure:

for example-

```
struct xyz
{
 int array[100];
};
```

```
int somefunc(struct xyz large)
{
 ...
}
void anotherfunc(void)
{
 struct xyz a;
 printf("%d\n", somefunc(a));
}
#####
#####
```

Que- What is the O/P of following program?

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

```
main()
{
char arr[]="string";

printf("%c",*arr);

arr++;

printf("%c",*arr);

}
```

Ans- Error !!!

error Line 9 : '++' needs l-value

A side effect of arr++ is an attempt to change the value of arr.we can't use statement like , 'arr++' because , arr is a constant pointer to the allocated memory

```
#####
#####
```

Q.1 predict the output for this code snippet with reason?

```
main()
{
int (*ptr)[5];
```

```
char *p;
```

```
int arr[5]={1,2,3,4,5};
ptr=arr;
p=arr;
```

```
printf("ptr=%u \n p=%u",ptr,p);
p++;
ptr++;
printf("ptr=%u \n p=%u",ptr,p);

printf("*p=%u\n *ptr=%u",*p,*ptr);

printf("**ptr=%u\n",**ptr);

}
```

A.1 ptr=2402524240 p=2402524240

ptr=2402524260 p=2402524241 ( ptr is pointer to array so increments by 20 bytes  
int size x array size , while p is pointer to char so increments by 1 byte)

\*p=0 ( p is char pointer so it takes first byte of 4 bytes assigned that is zero here  
0000 0000 0000 0001 , value can be diff if it is big endian)  
\*ptr=2402524260 ( single dereference cant give u value as it is pointer to array,we  
have to double dereference it for getting value)

\*\*ptr= 0; ( it gives value to next array memory is not assigned so it can take any value)

```
#####
#####
```

Que- X % Y (X modulus Y )

Given- y is a power of 2.

calculate X % Y without using % operator.

Solution-

$$x \text{ modulo } y = (x \& (y - 1))$$

```
#####
#####
```

Que- int a= some bit stream;

int b = another bit stream;

only 1 bit difference between 'a' and 'b'. find out the position of mismatch bit.

Ans-

```
#include <iostream>
using namespace std;
int main()
{
 int a=1123,b=1091; // declared a and b with integer and difference of their bits is only 1
 int c,d;
 int count=0;
 c= a ^ b ; //
 d= c-1;
 if((c & d) ==0)
 {
 while(c !=0)
 {
 count++;
 c=(c>>1);
 }
 }
 cout<<count; // return position where bits are mismatched
}
```

```
#####
#####
```

Que-

I have a bit stream and without using for loop, I want to set m bit to n bit.

(m >n)

eg:

Bitstream = 10 10 10 10 01 00

m= 10

n=4,

O/P- 00 01 11 11 00 00

Solution :

$(-1 \ll n) \& ((1 \ll m) - 1)$

Note: given solution set all the bit within the range but turn off the bits which are beyond the range.

to overcome this problem we can use this logic.

num | (( (1<<( m- n+1 )-1 ) << ( n-1 ) )

```
#####
#####
```

strtok

char \*strtok(char \*str, const char \*delim);

The strtok() function parses a string into a sequence of tokens. On the first call to strtok() the string to be parsed should be specified in str. In each subsequent call that should parse the same string, str should be NULL. The delim argument specifies a set of bytes that delimit the tokens in the parsed string. The caller may specify different strings in delim in successive calls that parse the same string. Each call to

`strtok()` returns a pointer to a null-terminated string containing the next token. This string does not include the delimiting byte. If no more tokens are found, `strtok()` returns `NULL`.

```
#include<string.h>

int main(){
char str[]="hello-this is rider";
char *pch;
char* a[10];
int i=0,j;
pch=strtok(str," "); //we can add more special character

while(pch!=NULL){
printf("%s\n",pch);
a[i]=pch;
i++;
pch=strtok(NULL," ");
}
reverseWords(a,i);
printf("Reverse:\n");
for(j=0;j<i;j++)
printf("%s\n", a[j]);
return 0;
}
```

O/P:

```
rider
is
hello-this
```

---



---

```
void main()
{
int a[2][3][2] = {{{2,4},{7,8},{3,4}},{ {2,2},{2,3},{3,4}}};
printf("0x%u 0x%u 0x%u %d \n",a,*a,**a,***a);
```

```
printf("0x%u 0x%u 0x%u %d \n", a+1, *a+1, **a+1, ***a+1);
}
```

O/P:

```
0x382857360 0x382857360 0x382857360 2
0x382857384 0x382857368 0x382857364 3
```

Explanation:

The base address of array is 0x382857360.

As we know element at location [1][1][1] can be accessed as "a[1][1][1]" or \*( \*( \*(a + 1) + 1) + 1).

T \*p; // p is a pointer to an object of type T

When a pointer p is pointing to an object of type T, the expression \*p is of type T.  
For example a is of type array of 2 two dimensional arrays.

The type of the expression \*a is "array of arrays (i.e. two dimensional array)".

Based on the above concept translating the expression \*( \*( \*(buffer + 1) + 1) + 1).

For a three-dimensional array, reference to an array has type "pointer to array of 3 ints, of which each element as an array of 2 ints".

Therefore, a+1 is pointing to the memory location of first element of the second 2D array, a.

So,

a - An array of 2 two dimensional arrays, i.e. its type is "array of 2 two dimensional arrays".

a + 1 - displacement for 2nd element in the array of 5 two dimensional arrays.

i.e a+1 is equivalent to a[1][0][0]

Hence  $382857360 + 3 * (2 \text{ ints} * 4 \text{ bytes}) = 382857384$

Similarly, for \*a+1

$*(a + 0) + 1$  - displacement to access 1th element in the array of 3 one dimensional arrays.

Therefore \*a+1 is equivalent to a[0][1][0]

Similarly, for  $**a+1$

$*( *(a + 0) + 0) + 1$  - displacement to get element at 1th position in the single dimension array of integers.

Therefore  $**a+1$  is equivalent to  $a[0][0][1]$

Similarly, for  $***a+1$  is equivalent to value at  $a[0][0][1]$

$*( *( *(a + 0) + 0) + 0)$  - accessing the element at 0th position (the overall expression type is int now).

### String Literals

```
char *string = "English"; /* string literal */
char array[] = "English"; /* Char array not a string literal */
array[0] = 'V'; /*Legal, no error*/
string[0] = 'V'; /* Undefined */
```

ANSI C standard defines String literals as " A string has type ``array of characters'' and storage class static and is initialized with the given characters. Whether identical string literals are distinct is implementation defined, and the behavior of a program that attempts to alter a string literal is undefined." Trying to modify string literal will give segmentation fault on Linux however it may work on other OS or on processors which do not have memory protection.

In gcc/Linux string literal [char \*ptr] is stored read-only area hence you get a segmentation fault. But if non gcc compiler stores string literal in Read-Write area OR if the processor does not have MMU and OS does not offer memory protection this program will work without any error. This is because C language standard does not clearly define what should be result of code trying to modify string literals; C standard had left it to compilers/OS/Processor to use in best way they can.

Note that

### Structure alignment

1. Memory is always byte addressable.
2. For speed optimization compiler by default uses alignment which is same as data bus size. For ex. In case of machine with 32-bit data bus alignment is done in multiples of 4-bytes.
3. In case of data types smaller than size of data bus padding bytes are added so that each variable is aligned.
4. Problem with padding: In real communication and in storage padding is never used as

it is unnecessary wastage of time & space. But if the padded structures used to reference protocol header or in media files data it could lead to incorrect results because of difference in alignment of actual data verses padded structures.

5. To avoid such issues compilers provide `#pragma pack` [see compiler document for specific syntax]. With this pragma programmer can instruct compiler not to add any padding and all the structures must be packed even though it mean compromise of efficiency.

### Precedence & Associativity

```
#include<stdio.h>
main()
{
int a =10, b=20, c=0;
c=a+++b; /*No spaces in between*/
printf("a=%d, b=%d, c=%d\n",a,b,c);

return 0;
}
```

ANS:a=11, b=20 c=30

1. This is because ++ has higher precedence than +
2. ++ has "Right to Left" associativity hence it gets associated with a and becomes a++ i.e. post increment.

**Q1.) Reverse a string using bitwise operator?**

Ans: using xor operator.

```
Int main(){
Char x[] = "Manohar";
Char *string = x;
int len = strlen(string);
for (int i=0; i<len/2; i++){
 x[len-i-1] ^= x[i];
 x[i] ^= x[len-i-1];
 x[len-i-1] ^= x[i];
}
}
```

**Q2.) printf("%d", -1 <<4); what is the output ?**

Ans: 1 is represented in binary form is 0000 0000 0000 0001

-1 means 1's compliment                    1111 1111 1111 1110

Output is FFF0

**Q3.) Use of bit fields in C?**

Ans: Both C and C++ allow integer members to be stored into memory spaces smaller than the compiler would ordinarily allow. Width in bit fields explicitly declared and A bit field declaration may not use either of the type qualifiers, const or volatile.

**The following restrictions apply to bit fields. You cannot:**

- Define an array of bit fields
- Take the address of a bit field
- Have a pointer to a bit field
- Have a reference to a bit field

The following example demonstrates padding. The structure kitchen contains eight members totaling 16 bytes.

```
struct on_off {
 unsigned light : 1;
 unsigned toaster : 1;
 int count; /* 4 bytes */
 unsigned ac : 4;
 unsigned : 4;
 unsigned clock : 1;
 unsigned : 0;
 unsigned flag : 1;
} kitchen ;
```

| Member Name          | Storage Occupied                         |
|----------------------|------------------------------------------|
| Light                | 1 bit                                    |
| Toaster              | 1 bit                                    |
| (padding -- 30 bits) | To the next int boundary                 |
| Count                | The size of an int (4 bytes)             |
| Ac                   | 4 bits                                   |
| (unnamed field)      | 4 bits                                   |
| Clock                | 1 bit                                    |
| (padding -- 23 bits) | To the next int boundary (unnamed field) |
| Flag                 | 1 bit                                    |
| (padding -- 31 bits) | To the next int boundary                 |

**Q4.) what is the output of given code ?**

Int x = 5;

Printf("%d %d %d",x, x<<2, x>>2);

Ans: 5 20 1

**Q5.) char \*ptr=NULL;**

Char arr[10]={"abcd";

Strcpy(ptr,arr);

Printf("%s",ptr);

What is the output?

Ans: segmentation fault. Since pointer is not pointing to any valid memory.

Q6.) Int x[]={1,2,3,4,5};

Int \*ptr, \*\*ptr2;

Ptr = x;

Ptr2 = &ptr ;

How do you update x[2]= 10 using ptr2 ?

Ans: ptr2 + 2 and \*\*ptr = 10;

Or

\*\*(ptr2 + 2) = 10;

Q7.) Convert the expression ((A + B) \* C – (D – E) ^ (F + G)) to postfix notation.

Ans: AB + C \* DE - - FG + ^

Q8.) Predict the output from the given code?

Struct abc{

    int a;

    float b;

}d;

main(){

d.a = 1;

printf("%f" , d.b);

}

Ans: 0.0

Q9.) Predict the which one is not valid statement?

1.++d++++e++;

2.f\*=g+=h=5;

3.L-- >> M << -- N;

4.int a(int a), b = 0, c =a((c=b,++b));

5.i- >j< -k

Ans: 5

Q4:

int p1 = sizeof("HELLO");

Int p2 = strlen("HELLO");

printf("\n p1 : %d p2: %d \n",p1,p2);

Output :

P1 = 6;

P2 = 5 ;

If "char \*str = "smartplay";int p1 = sizeof(str);" then p1 is 8. Pointer size 8 will be return.

| Q.No | Question | Output | Discussion |
|------|----------|--------|------------|
|------|----------|--------|------------|

|                               |                                                                                                                                                                                  |                                                                                                                                                    |                                                                                                                                                       |
|-------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1                             | char *const ptr;<br>char ch1,ch2;<br>ptr = &ch1;<br>ptr = &ch2;                                                                                                                  | Compilation Error:<br>day1.c:6:1: error: assignment<br>of read-only variable ‘ptr’<br>day1.c:7:1: error: assignment<br>of read-only variable ‘ptr’ | char *const ptr; ==> Address is<br>constant , value can be changed.<br>Char const *ptr; ==> Value is<br>constant , address can be changed.            |
| What is value stored in ptr ? |                                                                                                                                                                                  |                                                                                                                                                    |                                                                                                                                                       |
| 2                             | char arr[7] = "HELLO123";<br>printf("\n arr : %s",arr);<br>printf("\n arr2 %c \n",*arr);                                                                                         | arr : HELLO12<br>arr2 : H                                                                                                                          |                                                                                                                                                       |
| 3                             | Explain situation to use double<br>pointer                                                                                                                                       | int *p;<br>func(&p);<br><br>void func(int **q){<br>*q = malloc(sizeof(int));<br>}                                                                  |                                                                                                                                                       |
| 4                             | What the value of p1 & p2;<br>char *str = "smartplay";<br>int p1 = sizeof(str);<br>Int p2 = strlen(str);<br><br>printf("\n p1 : %d p2: %d<br>\n",p1,p2);                         | P1 = 6;<br>P2 = 5 ;                                                                                                                                | Sizeof operator returns the size of<br>string including null character<br><br>Strlen function returns length of a<br>string excluding null character. |
| 5                             | char *str1 = "smartplay";<br>char *str2[] = "smartplay";<br><br>printf("\n str1 : %u str2: %u<br>\n",sizeof(str1),sizeof(str2));                                                 | Compilation error at *str2[].<br>error: invalid initializer                                                                                        | str2 pointer should be memory<br>allocated.<br><br>Char *str2 = (char *)<br>malloc(sizeof(5));<br><br>Do dynamic allocation to avoid<br>such errors.  |
| 6                             | #define TRUE 0<br>#define FALSE -1<br>#define NULL 0<br><br>void main(){<br><br>if(TRUE) printf("TRUE");<br><br>else if(FALSE) printf("FALSE");<br><br>Else printf("NULL");<br>} | FALSE                                                                                                                                              | -1 = FF Consider as TRUE                                                                                                                              |

|   |                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |                                                                                                             |                                                                                                                                                                                                                         |
|---|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 7 | <pre>enum actor {     abc = 5,     xyz = -2,     gcd,     hcf };  void main(){      enum actor a =0;     switch(a) {         case abc:             printf("abc");             break;         case xyz:             printf("xyz");             break;         case gcd:             printf("gcd");             break;         case hcf:             printf("hcf");             break;         default:             printf("default");             break;     } }</pre> | hcf                                                                                                         | xyz = -2,<br>gcd, // gets -1 (xyz + 1)<br>hcf // gets 0 ( gcd +1)                                                                                                                                                       |
| 8 | <pre>switch(*(1 + "AB" "CD" + 1)) {     case 'A':         printf("A");         break;     case 'B':         printf("B");         break;     case 'C':         printf("C");         break;     case 'D':         printf("D");         break;     default:         printf("default");         break; }</pre>                                                                                                                                                            | C                                                                                                           | 1 +ABCD + 1<br>2 + ABCD<br><br>A = 0 , B = 1 C = 2 , D = 3<br><br>Expected value is 2 , So it reach case C:<br><br>If switch((1 + "AB" "CD" + 1)) then its an compilation error : error: switch quantity not an integer |
| 9 | <pre>struct student {     int rno;     char name[10]; }st;  void main(){      st.rno = 10;     st.name = "hello"; }</pre>                                                                                                                                                                                                                                                                                                                                             | Compilation Error :<br>error: incompatible types<br>when assigning to type<br>'char[10]' from type 'char *' | st.name = "hello"; is not allowed<br>strcpy(st.name,"hello"); is allowed                                                                                                                                                |

|    |                                                                                             |                                                                                                                                                       |                         |
|----|---------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------|
| 10 | <pre>int xx = NULL; int yy = 10; int zz = 0;  zz = xx + yy;  printf("Z : %d \n ",zz);</pre> | Z = 10;                                                                                                                                               | NULL Considered as a 0. |
| 11 | <pre>printf("char : %c ",59); printf("int : %d ",59); printf("string : %s ",59);</pre>      | Char - ; Int - 59 String - Segmentation fault                                                                                                         |                         |
| 12 | <pre>int i = 4,5,6; Int l = (4,5,6);</pre>                                                  | First statement - Not allowed<br>- error: expected identifier or<br>'(' before numeric constant.<br><br>Second statement - l = 6 will<br>be assigned. |                         |
| 13 | <pre>int l = -1; while(+(+i) != 0) { ++i; }</pre>                                           | l = 0;                                                                                                                                                |                         |

sorry for wrong answer in 1st question. please check it.

Q 1) if else statement is better than switch case if yes then why?

switch statement is better than if-else, because compiler is better in optimizing a switch-statement than an if-statement.

The compiler doesn't know if the order of evaluating the if-statements is important to you, and can't perform any optimizations there. You could be calling methods in the if-statements, influencing variables. With the switch-statement it knows that all clauses can be evaluated at the same time and can put them in whatever order is most efficient.

Q 1) if else statement is better than switch case if yes then why?

If else statement is better than switch, because compiler is better in optimizing a switch-statement than an if-statement.

The compiler doesn't know if the order of evaluating the if-statements is important to you, and can't perform any optimizations there. You could be calling methods in the if-statements, influencing variables. With the switch-statement it knows that all clauses can be evaluated at the same time and can put them in whatever order is most efficient.

Q 2)

```
main()
{
int i=5;
change();
int i=10;
printf("%d",i)
}
change()
{
?(need to write code here)
}
```

in above statement print the 5 with respect to the change function.

Ans. #define printf(a,b) print("5");

Description:

defined in above of main function is also It Prints 5.

Printf function is a Built in function Even it can be used as a User defined function.

Q 3)

```
#define printd(expr) printf(#expr " %d",expr)
main()
{
int a=10,b=2;
printd(a/b);
}
```

What is o/p?

Ans: a/b=5;  
#expr treats that variable names.

Q.1 write a function which takes 7 as input and returns 4 and vice versa ?

```
A.1 int func1(int p) // input p should be 7 or 4
{
int i;
i=(p^3);
return i;
}
```

Q.2 what is stack flow after the statement being executed ?

OR

How these four things gets into picture 1. Compiler 2. C Language 3. Glibc 4. Kernel after the below statement?

```
char *p=((char *)malloc (10));
```

A.2

1. C Language :- It specifies the header file <stdlib.h> ,which will check the prototype.

2. Compiler :- It takes the output of the preprocessor, and the source code, and generates assembler source code it is kept in an object file.

3. Glibc :- The glibc package contains standard libraries.In order to save disk space and memory, as well as to make upgrading easier, common system code is kept in one place and shared between programs. This particular package contains the most important sets of shared libraries: the standard C library and the standard math library. Without these two libraries, a Linux system will not function

So object file generated now gets linked with the Glibc where in implementation of malloc is being defined.

There are other APIs like uclibc , bionic libc which can be linked with different OS (Linux, BCD, QNX) in .Config file while building kernel, depending on the requirement of the user.

3. Kernel :- Glibc calls the kernel via brk() system call.

1.Allocated 2D memory with malloc.

Ans: Pointers can be easily used to create a 2D array in C using malloc. The idea is to first create a one dimensional array of pointers, and then, for each array entry, create another one dimensional array. Here's a sample code:

```
double**theArray;
theArray=(double**)malloc(arraySizeX*sizeof(double*));
for(int i=0; i<arraySizeX;i++)
theArray[i]=(double*)malloc(arraySizeY*sizeof(double));
```

-----  
2.

```
int * fun()
{
static int i;
i=20;
return (&i);
}
```

Ans: its depend on OS implementation because static variable store in Data segment but it will try to

return the value from stack so some times it works some time it does not work because it is dependent on how OS have property of memory distribution.

3.

```
int fun(int x)
{
 int y;
 y=x-1;
 return((x&y)==0);
}
```

Ans: if value is 2 power n then it will return 1 else return 0 ;

sol: case 1: x=4 and then y=4-1=3

x=0100 and y =0011

So ((0100 & 0011)==0) it is true it will return 1.

Case2: x=6 and the y=5

x=0110 and y=0101

So ((0110 & 0101)==0) it is not true so it will return 0.

conclusion: if value is 2 power n then it will return 1 else return 0 ;

Que- How can we pass entire array (all element of array) to the function.

Ans- for passing an array, wrap it in a structure:

for example-

```
struct xyz
{
 int array[100];
};

int somefunc(struct xyz large)
{
 ...
}
void anotherfunc(void)
{
 struct xyz a;
 printf("%d\n", somefunc(a));
}
#####
#####
```

Que- What is the O/P of following program?

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

```
main()
{
char arr[]="string";
printf("%c",*arr);
arr++;
printf("%c",*arr);
}
```

Ans- Error !!!  
error Line 9 : '++' needs l-value

A side effect of arr++ is an attempt to change the value of arr.we can't use statement like , 'arr++'  
because , arr is a constant pointer to the allocated memory

```
#####
#####
```

Q.1 predict the output for this code snippet with reason?

```
main()
{
int (*ptr)[5];
```

```
char *p;
```

```
int arr[5]={1,2,3,4,5};
ptr=arr;
p=arr;
printf("ptr=%u \n p=%u",ptr,p);
p++;
ptr++;
printf("ptr=%u \n p=%u",ptr,p);
```

```
printf("*p=%u\n *ptr=%u",*p,*ptr);
```

```
printf("**ptr=%u\n",**ptr);
```

```
}
```

A.1 ptr=2402524240 p=2402524240

ptr=2402524260 p=2402524241 ( ptr is pointer to array so increments by 20 bytes int size x array size , while p is pointer to char so increments by 1 byte)

\*p=0 ( p is char pointer so it takes first byte of 4 bytes assigned that is zero here 0000 0000 0000 0001 , value can be diff if it is big endian)

\*ptr=2402524260 ( single dereference cant give u value as it is pointer to array,we have to double dereference it for getting value)

\*\*ptr= 0; ( it gives value to next array memory is not assigned so it can take any value)

```
#####
```

Que- X % Y (X modulus Y )

Given- y is a power of 2.

calculate X % Y without using % operator.

Solution-

$x \text{ modulo } y = (x \& (y - 1))$

```
#####
```

Que- int a= some bit stream;

int b = another bit stream;

only 1 bit difference between 'a' and 'b'. find out the position of mismatch bit.

Ans-

```
#include <iostream>
using namespace std;
int main()
{
int a=1123,b=1091; // declared a and b with integer and difference of their bits is only 1
int c,d;
int count=0;
c= a ^ b ; //
d= c-1;
if((c & d) ==0)
{
while(c !=0)
{
count++;
c=(c>>1);
}
}
cout<<count; // return position where bits are mismatched
}
```

```
#####
```

Que-

I have a bit stream and without using for loop, I want to set m bit to n bit.

(m >n)  
 eg:  
 Bitstream = 10 10 10 10 01 00  
 m= 10  
 n=4,  
 O/P- 00 01 11 11 00 00

Solution :  
 $(-1 \ll n) \& ((1 \ll m) - 1)$

Note: given solution set all the bit within the range but turn off the bits which are beyond the range.  
 to overcome this problem we can use this logic.

`num | ( ( (1<<( m- n+1 )-1 ) << ( n-1 ) )`

#####

strtok

`char *strtok(char *str, const char *delim);`

The strtok() function parses a string into a sequence of tokens. On the first call to strtok() the string to be parsed should be specified in str. In each subsequent call that should parse the same string, str should be NULL. The delim argument specifies a set of bytes that delimit the tokens in the parsed string. The caller may specify different strings in delim in successive calls that parse the same string. Each call to strtok() returns a pointer to a null-terminated string containing the next token. This string does not include the delimiting byte. If no more tokens are found, strtok() returns NULL.

```
#include<string.h>

int main(){
char str[]="hello-this is rider";
char *pch;
char* a[10];
int i=0,j;
pch=strtok(str," "); //we can add more special character

while(pch!=NULL){
printf("%s\n",pch);
a[i]=pch;
i++;
pch=strtok(NULL," ");
}
}
```

```
reverseWords(a,i);
printf("Reverse:\n");

for(j=0;j<i;j++)
printf("%s\n", a[j]);
return 0;
}
```

O/P:  
rider  
is  
hello-this

---

```
void main()
{
int a[2][3][2] = {{{2,4},{7,8},{3,4}},{ {2,2},{2,3},{3,4}}};
printf("0x%u 0x%u 0x%u %d \n",a,*a,**a,***a);
printf("0x%u 0x%u 0x%u %d \n",a+1,*a+1,**a+1,***a+1);
}
```

O/P:  
0x382857360 0x382857360 0x382857360 2  
0x382857384 0x382857368 0x382857364 3

Explanation:

The base address of array is 0x382857360.

As we know element at location [1][1][1] can be accessed as “a[1][1][1]” or \*( \*( \*(a + 1) + 1) + 1).

T \*p; // p is a pointer to an object of type T

When a pointer p is pointing to an object of type T, the expression \*p is of type T.

For example a is of type array of 2 two dimensional arrays.

The type of the expression \*a is “array of arrays (i.e. two dimensional array)”.

Based on the above concept translating the expression \*( \*( \*(buffer + 1) + 1) + 1).

For a three-dimensional array, reference to an array has type “pointer to array of 3 ints, of which each element as an array of 2 ints”.

Therefore, a+1 is pointing to the memory location of first element of the second 2D array, a.

So,

a - An array of 2 two dimensional arrays, i.e. its type is “array of 2 two dimensional arrays”.

a + 1 - displacement for 2nd element in the array of 5 two dimensional arrays.

i.e a+1 is equivalent to a[1][0][0]

Hence  $382857360 + 3 * (2 \text{ ints} * 4 \text{ bytes}) = 382857384$

Similarly, for \*a+1

$*(a + 0) + 1$  - displacement to access 1th element in the array of 3 one dimensional arrays.

Therefore \*a+1 is equivalent to a[0][1][0]

Similarly, for \*\*a+1

$*(*(a + 0) + 0) + 1$  - displacement to get element at 1th position in the single dimension array of integers.

Therefore \*\*a+1 is equivalent to a[0][0][1]

Similarly, for \*\*\*a+1 is equivalent to value at a[0][0][1]

$*(*(*(a + 0) + 0) + 0)$  - accessing the element at 0th position (the overall expression type is int now).

### String Literals

```
char *string = "English"; /* string literal */
char array[] = "English"; /* Char array not a string literal */
array[0] = 'V' ; /*Legal, no error*/
string[0] = 'V'; /* Undefined */
```

ANSI C standard defines String literals as " A string has type ``array of characters" and storage class static and is initialized with the given characters. Whether identical string literals are distinct is implementation defined, and the behavior of a program that attempts to alter a string literal is undefined." Trying to modify string literal will give segmentation fault on Linux however it may work on other OS or on processors which do not have memory protection.

In gcc/Linux string literal [char \*ptr] is stored read-only area hence you get a segmentation fault. But if non gcc compiler stores string literal in Read-Write area OR if the processor does not have MMU and OS does not offer memory protection this program will work without any error. This is because C language standard does not clearly define what should be result of code trying to modify string literals; C standard had left it to compilers/OS/Processor to use in best way they can.

Note that

### Structure alignment

1. Memory is always byte addressable.
2. For speed optimization compiler by default uses alignment which is same as data bus size. For ex. In case of machine with 32-bit data bus alignment is done in multiples of 4-bytes.
3. In case of data types smaller than size of data bus padding bytes are added so that each variable is aligned.
4. Problem with padding: In real communication and in storage padding is never used as it is unnecessary wastage of time & space. But if the padded structures used to reference protocol header or in media

files data it could lead to incorrect results because of difference in alignment of actual data versus padded structures.

5. To avoid such issues compilers provide #pragma pack [see compiler document for specific syntax]. With this pragma programmer can instruct compiler not to add any padding and all the structures must be packed even though it mean compromise of efficiency.

#### Precedence & Associativity

```
#include<stdio.h>
main()
{
int a =10, b=20, c=0;
c=a+++b; /*No spaces in between*/
printf("a=%d, b=%d, c=%d\n",a,b,c);

return 0;
}
```

ANS:a=11, b=20 c=30

1. This is because ++ has higher precedence than +
  2. ++ has “Right to Left” associativity hence it gets associated with a and becomes a++ i.e. post increment.
- 

Below are the list of questions discussed on C-session (Apr 16 & 17).

| Q.No | Question                                                                                                                                                 | Output                                                                                                                                             | Discussion                                                                                                                                      |
|------|----------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------|
| 1    | char *const ptr;<br>char ch1,ch2;<br>ptr = &ch1;<br>ptr = &ch2;<br><br>What is value stored in ptr ?                                                     | Compilation Error:<br>day1.c:6:1: error: assignment<br>of read-only variable ‘ptr’<br>day1.c:7:1: error: assignment<br>of read-only variable ‘ptr’ | char *const ptr; ==> Address is constant , value can be changed.<br>Char const *ptr; ==> Value is constant , address can be changed.            |
| 2    | char arr[7] = “HELLO123”;<br>printf(“\n arr : %s”,arr);<br>printf(“\n arr2 %c \n”,*arr);                                                                 | arr : HELLO12<br>arr2 : H                                                                                                                          |                                                                                                                                                 |
| 3    | Explain situation to use double pointer                                                                                                                  | int *p;<br>func(&p);<br><br>void func(int **q){<br>*q = malloc(sizeof(int));<br>}                                                                  |                                                                                                                                                 |
| 4    | What the value of p1 & p2;<br>char *str = "smartplay";<br>int p1 = sizeof(str);<br>Int p2 = strlen(str);<br><br>printf("\n p1 : %d p2: %d<br>\n",p1,p2); | P1 = 6;<br>P2 = 5 ;                                                                                                                                | Sizeof operator returns the size of string including null character<br><br>Strlen function returns length of a string excluding null character. |

|   |                                                                                                                                                                                                                                                                                                                                        |                                                             |                                                                                                                                             |
|---|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------|
| 5 | char *str1 = "smartplay";<br>char *str2[] = "smartplay";<br><br>printf("\n str1 : %u str2: %u<br>\n",sizeof(str1),sizeof(str2));                                                                                                                                                                                                       | Compilation error at *str2[].<br>error: invalid initializer | str2 pointer should be memory allocated.<br><br>Char *str2 = (char *) malloc(sizeof(5));<br><br>Do dynamic allocation to avoid such errors. |
| 6 | #define TRUE 0<br>#define FALSE -1<br>#define NULL 0<br><br>void main(){<br><br>if(TRUE) printf("TRUE");<br><br>else if(FALSE) printf("FALSE");<br><br>Else printf("NULL");<br>}                                                                                                                                                       | FALSE                                                       | -1 = FF Consider as TRUE                                                                                                                    |
| 7 | enum actor {<br>abc = 5,<br>xyz = -2,<br>gcd,<br>hcf<br>};<br>void main(){<br><br>enum actor a =0;<br>switch(a) {<br>case abc:<br>printf("abc");<br>break;<br>case xyz:<br>printf("xyz");<br>break;<br>case gcd:<br>printf("gcd");<br>break;<br>case hcf:<br>printf("hcf");<br>break;<br>default:<br>printf("default");<br>break;<br>} | hcf                                                         | xyz = -2,<br>gcd, // gets -1 (xyz + 1)<br>hcf // gets 0 ( gcd +1)                                                                           |

|    |                                                                                                                                                                                                        |                                                                                                                                                     |                                                                                                                                                                                                                                 |
|----|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 8  | <pre>switch(*(1 + "AB" "CD" + 1)) { case 'A': printf("A"); break; case 'B': printf("B"); break; case 'C': printf("C"); break; case 'D': printf("D"); break; default: printf("default"); break; }</pre> | C                                                                                                                                                   | <p>1 +ABCD + 1<br/>2 + ABCD</p> <p>A = 0 , B = 1 C = 2 , D = 3</p> <p>Expected value is 2 , So it reach case C:</p> <p>If switch((1 + "AB" "CD" + 1)) then its an compilation error : error: switch quantity not an integer</p> |
| 9  | <pre>struct student { int rno; char name[10]; }st;  void main(){  st.rno = 10; st.name = "hello";</pre>                                                                                                | <p>Compilation Error :</p> <p>error: incompatible types when assigning to type 'char[10]' from type 'char *'</p>                                    | <p>st.name = "hello"; is not allowed</p> <p>strcpy(st.name,"hello"); is allowed</p>                                                                                                                                             |
| 10 | <pre>int xx = NULL; int yy = 10; int zz = 0;  zz = xx + yy;  printf("Z : %d \n ",zz);</pre>                                                                                                            | Z = 10;                                                                                                                                             | NULL Considered as a 0.                                                                                                                                                                                                         |
| 11 | <pre>printf("char : %c ",59); printf("int : %d ",59); printf("string : %s ",59);</pre>                                                                                                                 | <p>Char - ;</p> <p>Int - 59</p> <p>String - Segmentation fault</p>                                                                                  |                                                                                                                                                                                                                                 |
| 12 | <pre>int i = 4,5,6; Int l = (4,5,6);</pre>                                                                                                                                                             | <p>First statement - Not allowed - error: expected identifier or '(' before numeric constant.</p> <p>Second statement - l = 6 will be assigned.</p> |                                                                                                                                                                                                                                 |
| 13 | <pre>int l = -1; while(+(+i) != 0) { ++i; }</pre>                                                                                                                                                      | <p>I = 0;</p>                                                                                                                                       |                                                                                                                                                                                                                                 |

sorry for wrong answer in 1st question. please check it.

Q 1) if else statement is better than switch case if yes then why?

switch statement is better than if-else, because compiler is better in optimizing a switch-statement than an if-statement.

The compiler doesn't know if the order of evaluating the if-statements is important to you, and can't perform any optimizations there. You could be calling methods in the if-statements, influencing variables. With the switch-statement it knows that all clauses can be evaluated at the same time and can put them in whatever order is most efficient.

Q 1) if else statement is better than switch case if yes then why?

If else statement is better than switch, because compiler is better in optimizing a switch-statement than an if-statement.

The compiler doesn't know if the order of evaluating the if-statements is important to you, and can't perform any optimizations there. You could be calling methods in the if-statements, influencing variables. With the switch-statement it knows that all clauses can be evaluated at the same time and can put them in whatever order is most efficient.

Q 2)

```
main()
{
int i=5;
change();
int i=10;
printf("%d",i)
}
change()
{
?(need to write code here)
}
```

in above statement print the 5 with respect to the change function.

Ans. #define printf(a,b) print("5");

Description:

defined in above of main function is also It Prints 5.

Printf function is a Built in function Even it can be used as a User defined function.

Q 3)

```
#define printd(expr) printf(#expr “ %d”,expr)
main()
{
int a=10,b=2;
printd(a/b);
}
```

What is o/p?

Ans: a/b=5;  
#expr treats that variable names.

Q.1 write a function which takes 7 as input and returns 4 and vice versa ?

```
A.1 int func1(int p) // input p should be 7 or 4
{
int i;
i=(p^3);
return i;
}
```

Q.2 what is stack flow after the statement being executed ?

OR

How these four things gets into picture 1. Compiler 2. C Language 3. Glibc 4. Kernel after the below statement?

```
char *p=((char *)malloc (10));
```

A.2

1. C Language :- It specifies the header file <stdlib.h> ,which will check the prototype.

2. Compiler :- It takes the output of the preprocessor, and the source code, and generates assembler source code it is kept in an object file.

3. Glibc :- The glibc package contains standard libraries.In order to save disk space and memory, as well as to make upgrading easier, common system code is kept in one place and shared between programs. This particular package contains the most important sets of shared libraries: the standard C library and the standard math library. Without these two libraries, a Linux system will not function

So object file generated now gets linked with the Glibc where in implementation of malloc is being defined.

There are other APIs like uclibc , bionic libc which can be linked with different OS (Linux, BCD, QNX) in .Config file while building kernel, depending on the requirement of the user.

3. Kernel :- Glibc calls the kernel via brk() system call.

1. Allocated 2D memory with malloc.

Ans: Pointers can be easily used to create a 2D array in C using malloc. The idea is to first create a one dimensional array of pointers, and then, for each array entry, create another one dimensional array. Here's a sample code:

```
double**theArray;
theArray=(double**)malloc(arraySizeX*sizeof(double*));
for(int i=0; i<arraySizeX;i++)
theArray[i]=(double*)malloc(arraySizeY*sizeof(double));
```

---

2.

```
int * fun()
{
static int i;
i=20;
return (&i);
}
```

Ans: its depend on OS implementation because static variable store in Data segment but it will try to return the value from stack so some times it works some time it does not work because it is dependent on how OS have property of memory distribution.

3.

```
int fun(int x)
{
int y;
y=x-1;
return((x&y)==0);
}
```

Ans: if value is 2 power n then it will return 1 else return 0 ;

sol: case 1: x=4 and then y=4-1=3

x=0100 and y =0011

So ((0100 & 0011)==0) it is true it will return 1.

Case2: x=6 and the y=5

x=0110 and y=0101

So ( (0110 & 0101)==0) it is not true so it will return 0.

conclusion: if value is 2 power n then it will return 1 else return 0 ;

---

Que- How can we pass entire array (all element of array) to the function.

Ans- for passing an array, wrap it in a structure:

for example-

```
struct xyz
{
int array[100];
};
```

```
int somefunc(struct xyz large)
```

```
{
...
}
void anotherfunc(void)
{
struct xyz a;
printf("%d\n", somefunc(a));
}
#####
#####
```

Que- What is the O/P of following program?

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

```
main()
{
char arr[]="string";
printf("%c",*arr);

arr++;
printf("%c",*arr);

}
```

Ans- Error !!!  
error Line 9 : '++' needs l-value

A side effect of arr++ is an attempt to change the value of arr.we can't use statement like , 'arr++'  
because , arr is a constant pointer to the allocated memory

```
#####
#####
```

Q.1 predict the output for this code snippet with reason?

```
main()
{
int (*ptr)[5];
char *p;
```

```
int arr[5]={1,2,3,4,5};
ptr=arr;
p=arr;
```

```
printf("ptr=%u \n p=%u",ptr,p);
p++;
ptr++;
printf("ptr=%u \n p=%u",ptr,p);

printf("*p=%u\n *ptr=%u",*p,*ptr);

}

A.1 ptr=2402524240 p=2402524240

ptr=2402524260 p=2402524241 (ptr is pointer to array so increments by 20 bytes int size x array size ,
while p is pointer to char so increments by 1 byte)

*p=0 (p is char pointer so it takes first byte of 4 bytes assigned that is zero here
0000 0000 0000 0001 , value can be diff if it is big endian)
*ptr=2402524260 (single dereference cant give u value as it is pointer to array,we have to double
dereference it for getting value)

**ptr= 0; (it gives value to next array memory is not assigned so it can take any value)
```

#####  
Que- X % Y (X modulus Y )  
Given- y is a power of 2.

calculate X % Y without using % operator.

Solution-

$x \text{ modulo } y = (x \& (y - 1))$

#####  
Que- int a= some bit stream;  
int b = another bit stream;  
only 1 bit difference between 'a' and 'b'. find out the position of mismatch bit.  
Ans-

```
#include <iostream>
```

```
using namespace std;
int main()
{
int a=1123,b=1091; // declared a and b with integer and diffrence of their bits is only 1
int c,d;
int count=0;
c= a ^ b ; //
d= c-1;
if((c & d) ==0)
{
while(c !=0)
{
count++;
c=(c>>1);
}
}
cout<<count; // return position where bits are mismatched
}
```

```
#####
Que-
```

I have a bit stream and without using for loop, I want to set m bit to n bit.

(m >n)

eg:

Bitstream = 10 10 10 10 01 00

m= 10

n=4,

O/P- 00 01 11 11 00 00

Solution :

(-1 << n) & ((1<<m) - 1)

Note: given solution set all the bit within the range but turn off the bits which are beyond the range.  
to overcome this problem we can use this logic.

```
num | (((1<<(m- n+1)-1) << (n-1))
```

```
#####
strtok
```

```
char *strtok(char *str, const char *delim);
```

The strtok() function parses a string into a sequence of tokens. On the first call to strtok() the string to

be parsed should be specified in str. In each subsequent call that should parse the same string, str should be NULL. The delim argument specifies a set of bytes that delimit the tokens in the parsed string. The caller may specify different strings in delim in successive calls that parse the same string. Each call to strtok() returns a pointer to a null-terminated string containing the next token. This string does not include the delimiting byte. If no more tokens are found, strtok() returns NULL.

```
#include<string.h>

int main(){
char str[]="hello-this is rider";
char *pch;
char* a[10];
int i=0,j;
pch=strtok(str," "); //we can add more special character

while(pch!=NULL){
printf("%s\n",pch);
a[i]=pch;
i++;
pch=strtok(NULL," ");
}

reverseWords(a,i);
printf("Reverse:\n");

for(j=0;j<i;j++)
printf("%s\n", a[j]);
return 0;
}
```

O/P:  
rider  
is  
hello-this

---

```
void main()
{
int a[2][3][2] = {{{2,4},{7,8},{3,4}},{ {2,2},{2,3},{3,4}}};
printf("0x%u 0x%u 0x%u %d \n",a,*a,**a,***a);
printf("0x%u 0x%u 0x%u %d \n",a+1,*a+1,**a+1,***a+1);
}
```

O/P:  
0x382857360 0x382857360 0x382857360 2  
0x382857384 0x382857368 0x382857364 3

Explanation:

The base address of array is 0x382857360.

As we know element at location [1][1][1] can be accessed as “a[1][1][1]” or \*( \*( \*(a + 1) + 1) + 1).

T \*p; // p is a pointer to an object of type T

When a pointer p is pointing to an object of type T, the expression \*p is of type T.

For example a is of type array of 2 two dimensional arrays.

The type of the expression \*a is "array of arrays (i.e. two dimensional array)".

Based on the above concept translating the expression  $(*(\text{buffer} + 1) + 1) + 1$ .

For a three-dimensional array, reference to an array has type "pointer to array of 3 ints, of which each element as an array of 2 ints".

Therefore, a+1 is pointing to the memory location of first element of the second 2D array, a.

So,

a - An array of 2 two dimensional arrays, i.e. its type is "array of 2 two dimensional arrays".

a + 1 - displacement for 2nd element in the array of 5 two dimensional arrays.

i.e a+1 is equivalent to a[1][0][0]

Hence  $382857360 + 3 * (2 \text{ ints} * 4 \text{ bytes}) = 382857384$

Similarly, for \*a+1

$*(a + 0) + 1$  - displacement to access 1th element in the array of 3 one dimensional arrays.

Therefore \*a+1 is equivalent to a[0][1][0]

Similarly, for \*\*a+1

$*(*(a + 0) + 0) + 1$  - displacement to get element at 1th position in the single dimension array of integers.

Therefore \*\*a+1 is equivalent to a[0][0][1]

Similarly, for \*\*\*a+1 is equivalent to value at a[0][0][1]

$*(*(*(a + 0) + 0) + 0)$  - accessing the element at 0th position (the overall expression type is int now).

### String Literals

```
char *string = "English"; /* string literal */
char array[] = "English"; /* Char array not a string literal */
array[0] = 'V'; /*Legal, no error*/
string[0] = 'V'; /* Undefined */
```

ANSI C standard defines String literals as " A string has type ``array of characters" and storage class static and is initialized with the given characters. Whether identical string literals are distinct is implementation defined, and the behavior of a program that attempts to alter a string literal is undefined." Trying to modify string literal will give segmentation fault on Linux however it may work on other OS or on processors which do not have memory protection.

In gcc/Linux string literal [char \*ptr] is stored read-only area hence you get a segmentation fault. But if non gcc compiler stores string literal in Read-Write area OR if the processor does not have MMU and OS does not offer memory protection this program will work without any error. This is because C language standard does not clearly define what should be result of code trying to modify string literals; C standard had left it to compilers/OS/Processor to use in best way they can.

Note that

### Structure alignment

1. Memory is always byte addressable.
2. For speed optimization compiler by default uses alignment which is same as data bus size. For ex. In case of machine with 32-bit data bus alignment is done in multiples of 4-bytes.
3. In case of data types smaller than size of data bus padding bytes are added so that each variable is aligned.

4. Problem with padding: In real communication and in storage padding is never used as it is unnecessary wastage of time & space. But if the padded structures used to reference protocol header or in media files data it could lead to incorrect results because of difference in alignment of actual data versus padded structures.

5. To avoid such issues compilers provide #pragma pack [see compiler document for specific syntax]. With this pragma programmer can instruct compiler not to add any padding and all the structures must be packed even though it means compromise of efficiency.

#### Precedence & Associativity

```
#include<stdio.h>
main()
{
int a =10, b=20, c=0;
c=a+++b; /*No spaces in between*/
printf("a=%d, b=%d, c=%d\n",a,b,c);

return 0;
}
```

ANS:a=11, b=20 c=30

1. This is because ++ has higher precedence than +
  2. ++ has "Right to Left" associativity hence it gets associated with a and becomes a++ i.e. post increment.
- 

Thanks for a correction Venkat. Question updated.

Q4:

```
int p1 = sizeof("HELLO");
Int p2 = strlen("HELLO");

printf("\n p1 : %d p2: %d \n",p1,p2);
```

**Output :**

P1 = 6;  
P2 = 5 ;

If "char \*str = "smartplay";int p1 = sizeof(str);" then p1 is 8. Pointer size 8 will be returned.

| Q.No | Question | Output | Discussion |
|------|----------|--------|------------|
|------|----------|--------|------------|

|   |                                                                                                                                                                                      |                                                                                                                                                    |                                                                                                                                                       |
|---|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1 | char *const ptr;<br>char ch1,ch2;<br>ptr = &ch1;<br>ptr = &ch2;                                                                                                                      | Compilation Error:<br>day1.c:6:1: error: assignment<br>of read-only variable ‘ptr’<br>day1.c:7:1: error: assignment<br>of read-only variable ‘ptr’ | char *const ptr; ==> Address is<br>constant , value can be changed.<br>Char const *ptr; ==> Value is<br>constant , address can be changed.            |
|   | What is value stored in ptr ?                                                                                                                                                        |                                                                                                                                                    |                                                                                                                                                       |
| 2 | char arr[7] = "HELLO123";<br>printf("\n arr : %s",arr);<br>printf("\n arr2 %c \n",*arr);                                                                                             | arr : HELLO12<br>arr2 : H                                                                                                                          |                                                                                                                                                       |
| 3 | Explain situation to use double<br>pointer                                                                                                                                           | int *p;<br>func(&p);<br><br>void func(int **q){<br>*q = malloc(sizeof(int));<br>}                                                                  |                                                                                                                                                       |
| 4 | What the value of p1 & p2;<br>char *str = "smartplay";<br>int p1 = sizeof(str);<br>Int p2 = strlen(str);<br><br>printf("\n p1 : %d p2: %d<br>\n",p1,p2);                             | P1 = 6;<br>P2 = 5 ;                                                                                                                                | Sizeof operator returns the size of<br>string including null character<br><br>Strlen function returns length of a<br>string excluding null character. |
| 5 | char *str1 = "smartplay";<br>char *str2[] = "smartplay";<br><br>printf("\n str1 : %u str2: %u<br>\n",sizeof(str1),sizeof(str2));                                                     | Compilation error at *str2[].<br>error: invalid initializer                                                                                        | str2 pointer should be memory<br>allocated.<br><br>Char *str2 = (char *) malloc(sizeof(5));<br><br>Do dynamic allocation to avoid such<br>errors.     |
| 6 | #define TRUE 0<br>#define FALSE -1<br>#define NULL 0<br><br>void main(){<br><br>if(TRUE) printf("TRUE");<br><br>else if(FALSE) printf("FALSE");<br><br>Else printf("NULL");<br><br>} | FALSE                                                                                                                                              | -1 = FF Consider as TRUE                                                                                                                              |

|   |                                                                                                                                                                                                                                                                        |                                                                                                                 |                                                                                                                                                                                                                         |
|---|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 7 | <pre>enum actor { abc = 5, xyz = -2, gcd, hcf }; void main(){  enum actor a =0; switch(a) { case abc: printf("abc"); break; case xyz: printf("xyz"); break; case gcd: printf("gcd"); break; case hcf: printf("hcf"); break; default: printf("default"); break; }</pre> | hcf                                                                                                             | xyz = -2,<br>gcd, // gets -1 (xyz + 1)<br>hcf // gets 0 ( gcd +1)                                                                                                                                                       |
| 8 | <pre>switch(*(1 + "AB" "CD" + 1)) { case 'A': printf("A"); break; case 'B': printf("B"); break; case 'C': printf("C"); break; case 'D': printf("D"); break; default: printf("default"); break; }</pre>                                                                 | C                                                                                                               | 1 +ABCD + 1<br>2 + ABCD<br><br>A = 0 , B = 1 C = 2 , D = 3<br><br>Expected value is 2 , So it reach case C:<br><br>If switch((1 + "AB" "CD" + 1)) then its an compilation error : error: switch quantity not an integer |
| 9 | <pre>struct student { int rno; char name[10]; }st;  void main(){  st.rno = 10; st.name = "hello"; }</pre>                                                                                                                                                              | Compilation Error :<br><br>error: incompatible types<br>when assigning to type<br>‘char[10]’ from type ‘char *’ | st.name = "hello"; is not allowed<br><br>strcpy(st.name,"hello"); is allowed                                                                                                                                            |

|    |                                                                                             |                                                                                                                                        |                         |
|----|---------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------|-------------------------|
| 10 | <pre>int xx = NULL; int yy = 10; int zz = 0;  zz = xx + yy;  printf("Z : %d \n ",zz);</pre> | Z = 10;                                                                                                                                | NULL Considered as a 0. |
| 11 | <pre>printf("char : %c ",59); printf("int : %d ",59); printf("string : %s ",59);</pre>      | Char - ; Int - 59 String - Segmentation fault                                                                                          |                         |
| 12 | <pre>int i = 4,5,6; Int l = (4,5,6);</pre>                                                  | First statement - Not allowed - error: expected identifier or '(' before numeric constant.  Second statement - l = 6 will be assigned. |                         |
| 13 | <pre>int l = -1; while(+(+i) != 0) { ++i; }</pre>                                           | l = 0;                                                                                                                                 |                         |

sorry for wrong answer in 1st question. please check it.

Q 1) if else statement is better than switch case if yes then why?

switch statement is better than if-else, because compiler is better in optimizing a switch-statement than an if-statement.

The compiler doesn't know if the order of evaluating the if-statements is important to you, and can't perform any optimizations there. You could be calling methods in the if-statements, influencing variables. With the switch-statement it knows that all clauses can be evaluated at the same time and can put them in whatever order is most efficient.

Q 1) if else statement is better than switch case if yes then why?

If else statement is better than switch, because compiler is better in optimizing a switch-statement than an if-statement.

The compiler doesn't know if the order of evaluating the if-statements is important to you, and can't perform any optimizations there. You could be calling methods in the if-statements, influencing variables. With the switch-statement it knows that all clauses can be evaluated at the same time and can put them in whatever order is most efficient.

Q 2)

```
main()
{
int i=5;
change();
```

```
int i=10;
printf("%d",i)
}
change()
{
?(need to write code here)
}
```

in above statement print the 5 with respect to the change function.

Ans. #define printf(a,b) print("5");

Description:

defined in above of main function is also It Prints 5.

Printf function is a Built in function Even it can be used as a User defined function.

Q 3)

```
#define printd(expr) printf(#expr " %d",expr)
main()
{
int a=10,b=2;
printd(a/b);
}
```

What is o/p?

Ans: a/b=5;

#expr treats that variable names.

Q.1 write a function which takes 7 as input and returns 4 and vice versa ?

```
A.1 int func1(int p) // input p should be 7 or 4
{
int i;
i=(p^3);
return i;
}
```

Q.2 what is stack flow after the statement being executed ?

OR

How these four things gets into picture 1. Compiler 2. C Language 3. Glibc 4. Kernel after the below statement?

```
char *p=((char *)malloc (10));
```

A.2

1. C Language :- It specifies the header file <stdlib.h> ,which will check the prototype.

2. Compiler :- It takes the output of the preprocessor, and the source code, and generates assembler

source code it is kept in an object file.

3. Glibc :- The glibc package contains standard libraries. In order to save disk space and memory, as well as to make upgrading easier, common system code is kept in one place and shared between programs. This particular package contains the most important sets of shared libraries: the standard C library and the standard math library. Without these two libraries, a Linux system will not function

So object file generated now gets linked with the Glibc where in implementation of malloc is being defined.

There are other APIs like uclibc , bionic libc which can be linked with different OS (Linux, BCD, QNX) in .Config file while building kernel, depending on the requirement of the user.

3. Kernel :- Glibc calls the kernel via brk() system call.

1. Allocated 2D memory with malloc.

Ans: Pointers can be easily used to create a 2D array in C using malloc. The idea is to first create a one dimensional array of pointers, and then, for each array entry, create another one dimensional array. Here's a sample code:

```
double**theArray;
theArray=(double**)malloc(arraySizeX*sizeof(double*));
for(int i=0; i<arraySizeX;i++)
theArray[i]=(double*)malloc(arraySizeY*sizeof(double));
```

-----  
2.

```
int * fun()
{
static int i;
i=20;
return (&i);
}
```

Ans: its depend on OS implementation because static variable store in Data segment but it will try to return the value from stack so some times it works some time it does not work because it is dependent on how OS have property of memory distribution.

-----  
3.

```
int fun(int x)
{
int y;
y=x-1;
return((x&y)==0);
}
```

Ans: if value is 2 power n then it will return 1 else return 0 ;

sol: case 1: x=4 and then y=4-1=3

x=0100 and y =0011

So ((0100 & 0011)==0) it is true it will return 1.

Case2: x=6 and the y=5

x=0110 and y=0101

So ( (0110 & 0101)==0) it is not true so it will return 0.

Conclusion: if value is 2 power n then it will return 1 else return 0 ;

---

Que- How can we pass entire array (all element of array) to the function.

Ans- for passing an array, wrap it in a structure:

for example-

```
struct xyz
{
int array[100];
};

int somefunc(struct xyz large)
{
...
}
void anotherfunc(void)
{
struct xyz a;
printf("%d\n", somefunc(a));
}
#####
#####
```

Que- What is the O/P of following program?

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

```
main()
{
char arr[]="string";

printf("%c",*arr);

arr++;

printf("%c",*arr);

}
```

Ans- Error !!!  
error Line 9 : '++' needs l-value

A side effect of arr++ is an attempt to change the value of arr.we can't use statement like , 'arr++' because , arr is a constant pointer to the allocated memory

```
#####
#####
```

Q.1 predict the output for this code snippet with reason?

```
main()
{
 int (*ptr)[5];

 char *p;

 int arr[5]={1,2,3,4,5};
 ptr=arr;
 p=arr;
 printf("ptr=%u \n p=%u",ptr,p);
 p++;
 ptr++;
 printf("ptr=%u \n p=%u",ptr,p);

 printf("*p=%u\n *ptr=%u",*p,*ptr);

 printf("**ptr=%u\n",**ptr);

}
```

A.1 ptr=2402524240 p=2402524240

ptr=2402524260 p=2402524241 ( ptr is pointer to array so increments by 20 bytes int size x array size , while p is pointer to char so increments by 1 byte)

\*p=0 ( p is char pointer so it takes first byte of 4 bytes assigned that is zero here  
0000 0000 0000 0001 , value can be diff if it is big endian)  
\*ptr=2402524260 ( single dereference cant give u value as it is pointer to array,we have to double dereference it for getting value)

\*\*ptr= 0; ( it gives value to next array memory is not assigned so it can take any value)

```
#####
#####
```

Que- X % Y (X modulus Y )

Given- y is a power of 2.

calculate X % Y without using % operator.

Solution-

$x \text{ modulo } y = (x \& (y - 1))$

```
#####
```

Que- int a= some bit stream;  
int b = another bit stream;  
only 1 bit difference between 'a' and 'b'. find out the position of mismatch bit.

Ans-

```
#include <iostream>
using namespace std;
int main()
{
 int a=1123,b=1091; // declared a and b with integer and difference of their bits is only 1
 int c,d;
 int count=0;
 c= a ^ b ; //
 d= c-1;
 if((c & d) ==0)
 {
 while(c !=0)
 {
 count++;
 c=(c>>1);
 }
 }
 cout<<count; // return position where bits are mismatched
}
```

```
#####
```

Que-

I have a bit stream and without using for loop, I want to set m bit to n bit.

(m >n)

eg:

Bitstream = 10 10 10 10 01 00

m= 10

n=4,

O/P- 00 01 11 11 00 00

Solution :

$(-1 \ll n) \& ((1 \ll m) - 1)$

Note: given solution set all the bit within the range but turn off the bits which are beyond the range. to overcome this problem we can use this logic.

```
num | (((1<<(m-n+1))-1) << (n-1))
```

```
#####
```

`strtok`

```
char *strtok(char *str, const char *delim);
```

The `strtok()` function parses a string into a sequence of tokens. On the first call to `strtok()` the string to be parsed should be specified in `str`. In each subsequent call that should parse the same string, `str` should be `NULL`. The `delim` argument specifies a set of bytes that delimit the tokens in the parsed string. The caller may specify different strings in `delim` in successive calls that parse the same string. Each call to `strtok()` returns a pointer to a null-terminated string containing the next token. This string does not include the delimiting byte. If no more tokens are found, `strtok()` returns `NULL`.

```
#include<string.h>

int main(){
char str[]="hello-this is rider";
char *pch;
char* a[10];
int i=0,j;
pch=strtok(str," "); //we can add more special character

while(pch!=NULL){
printf("%s\n",pch);
a[i]=pch;
i++;
pch=strtok(NULL," ");
}

reverseWords(a,i);
printf("Reverse:\n");

for(j=0;j<i;j++)
printf("%s\n", a[j]);
return 0;
}
```

O/P:

```
rider
is
hello-this
```

---

```
void main()
{
int a[2][3][2] = {{{2,4},{7,8},{3,4}},{ {2,2},{2,3},{3,4}}};
printf("0x%u 0x%u 0x%u %d \n",a,*a,**a,***a);
printf("0x%u 0x%u 0x%u %d \n",a+1,*a+1,**a+1,***a+1);
}
```

O/P:  
0x382857360 0x382857360 0x382857360 2  
0x382857384 0x382857368 0x382857364 3

Explanation:

The base address of array is 0x382857360.

As we know element at location [1][1][1] can be accessed as “a[1][1][1]” or \*( \*( \*(a + 1) + 1) + 1).

T \*p; // p is a pointer to an object of type T

When a pointer p is pointing to an object of type T, the expression \*p is of type T.

For example a is of type array of 2 two dimensional arrays.

The type of the expression \*a is “array of arrays (i.e. two dimensional array)”.

Based on the above concept translating the expression \*( \*( \*(buffer + 1) + 1) + 1).

For a three-dimensional array, reference to an array has type "pointer to array of 3 ints, of which each element as an array of 2 ints".

Therefore, a+1 is pointing to the memory location of first element of the second 2D array, a.

So,

a - An array of 2 two dimensional arrays, i.e. its type is “array of 2 two dimensional arrays”.

a + 1 - displacement for 2nd element in the array of 5 two dimensional arrays.

i.e a+1 is equivalent to a[1][0][0]

Hence 382857360 + 3\*(2 ints \* 4 bytes) = 382857384

Similarly, for \*a+1

\*(a + 0) + 1 - displacement to access 1th element in the array of 3 one dimensional arrays.

Therefore \*a+1 is equivalent to a[0][1][0]

Similarly, for \*\*a+1

\*( \*(a + 0) + 0) + 1 - displacement to get element at 1th position in the single dimension array of integers.

Therefore \*\*a+1 is equivalent to a[0][0][1]

Similarly, for \*\*\*a+1 is equivalent to value at a[0][0][1]

\*( \*( \*(a + 0) + 0) + 0) - accessing the element at 0th position (the overall expression type is int now).

### String Literals

```
char *string = "English"; /* string literal */
char array[] = "English"; /* Char array not a string literal */
array[0] = 'V' ; /*Legal, no error*/
string[0] = 'V'; /* Undefined */
```

ANSI C standard defines String literals as " A string has type ``array of characters" and storage class static and is initialized with the given characters. Whether identical string literals are distinct is implementation defined, and the behavior of a program that attempts to alter a string literal is undefined." Trying to modify string literal will give segmentation fault on Linux however it may work on other OS or on processors which do not have memory protection.

In gcc/Linux string literal [char \*ptr] is stored read-only area hence you get a segmentation fault. But if non gcc compiler stores string literal in Read-Write area OR if the processor does not have MMU and OS does not offer memory protection this program will work without any error. This is because C language standard does not clearly define what should be result of code trying to modify string literals; C standard had left it to compilers/OS/Processor to use in best way they can.

Note that

### Structure alignment

1. Memory is always byte addressable.
2. For speed optimization compiler by default uses alignment which is same as data bus size. For ex. In case of machine with 32-bit data bus alignment is done in multiples of 4-bytes.
3. In case of data types smaller than size of data bus padding bytes are added so that each variable is aligned.
4. Problem with padding: In real communication and in storage padding is never used as it is unnecessary wastage of time & space. But if the padded structures used to reference protocol header or in media files data it could lead to incorrect results because of difference in alignment of actual data verses padded structures.
5. To avoid such issues compilers provide #pragma pack [see compiler document for specific syntax]. With this pragma programmer can instruct compiler not to add any padding and all the structures must be packed even though it mean compromise of efficiency.

### Precedence & Associativity

```
#include<stdio.h>
main()
{
int a =10, b=20, c=0;
c=a+++b; /*No spaces in between*/
printf("a=%d, b=%d, c=%d\n",a,b,c);

return 0;
}
```

ANS:a=11, b=20 c=30

1. This is because ++ has higher precedence than +
2. ++ has "Right to Left" associativity hence it gets associated with a and becomes a++ i.e. post increment.

---

## C Sessions

**Q1.) Reverse a string using bitwise operator?**

Ans: using xor operator.

```
Int main(){
Char x[] = "Manohar";
Char *string = x;
int len = strlen(string);
for (int i=0; i<len/2; i++){
 x[len-i-1] ^= x[i];
 x[i] ^= x[len-i-1];
 x[len-i-1] ^= x[i];
}
}
```

**Q2.) printf("%d", -1 <<4); what is the output ?**

Ans:1 is represented in binary form is 0000 0000 0000 0001

-1 means 1's compliment 1111 1111 1111 1110

Output is FFF0

**Q3.) Use of bit fields in C?**

Ans: Both C and C++ allow integer members to be stored into memory spaces smaller than the compiler would ordinarily

Width in bit fields explicitly declared and A bit field declaration may not use either of the type qualifiers, const or volatile

**The following restrictions apply to bit fields. You cannot:**

- Define an array of bit fields
- Take the address of a bit field
- Have a pointer to a bit field
- Have a reference to a bit field

The following example demonstrates padding. The structure kitchen contains eight members totaling 16 bytes.

```
struct on_off {
 unsigned light : 1;
 unsigned toaster : 1;
 int count; /* 4 bytes */
 unsigned ac : 4;
 unsigned : 4;
 unsigned clock : 1;
 unsigned : 0;
 unsigned flag : 1;
```

```
 } kitchen ;
```

| Member Name          | Storage Occupied                         |
|----------------------|------------------------------------------|
| Light                | 1 bit                                    |
| Toaster              | 1 bit                                    |
| (padding -- 30 bits) | To the next int boundary                 |
| Count                | The size of an int (4 bytes)             |
| Ac                   | 4 bits                                   |
| (unnamed field)      | 4 bits                                   |
| Clock                | 1 bit                                    |
| (padding -- 23 bits) | To the next int boundary (unnamed field) |
| Flag                 | 1 bit                                    |
| (padding -- 31 bits) | To the next int boundary                 |

**Q4.) what is the output of given code ?**

```
Int x = 5;
Printf("%d %d %d",x, x<<2, x>>2);
```

Ans: 5 20 1

**Q5.) char \*ptr=NULL;**

```
Char arr[10]={"abcd"};
```

```
Strcpy(ptr,arr);
```

```
Printf("%s",ptr);
```

**What is the output?**

Ans: segmentation fault. Since pointer is not pointing to any valid memory.

**Q6.) Int x[]={1,2,3,4,5};**

```
Int *ptr, **ptr2;
```

```
Ptr = x;
```

```
Ptr2 = &ptr ;
```

**How do you update x[2]= 10 using ptr2 ?**

Ans: ptr2 + 2 and \*\*ptr = 10;

Or

```
**(ptr2 + 2) = 10;
```

**Q7.) Convert the expression ((A + B) \* C – (D – E) ^ (F + G)) to postfix notation.**

Ans: AB + C \* DE - - FG + ^

**Q8.) Predict the output from the given code?**

```
Struct abc{
 int a;
 float b;
}d;
main(){
d.a = 1;
printf("%f" , d.b);
}
```

Ans: 0.0

**Q9.)Predict the which one is not valid statement?**

- 1.++d+++e++;
- 2.f\*=g+=h=5;
- 3.L-- >> M << -- N;
- 4.int a(int a), b = 0, c =a((c=b,++b));
- 5.i- >j< -k

Ans: 5

**Q4:**

```
int p1 = sizeof("HELLO");
Int p2 = strlen("HELLO");

printf("\n p1 : %d p2: %d \n",p1,p2);
```

**Output :**

P1 = 6;  
P2 = 5 ;

If "char \*str = "smartplay";int p1 = sizeof(str);" then p1 is 8. Pointer size 8 will be return.

Below are the list of questions discussed on C-session (Apr 16 & 17).

sorry for wrong answer in 1st question. please check it.

Q 1) if else statement is better than switch case if yes then why?

switch statement is better than if-else, because compiler is better in optimizing a switch-statement than an if-statement. The compiler doesn't know if the order of evaluating the if-statements is important to you, and can't perform any optimizations there. You could be calling methods in the if-statements, influencing variables.

With the switch-statement it knows that all clauses can be evaluated at the same time and can put them in whatever

order is most efficient.

Q 1) if else statement is better than switch case if yes then why?

If else statement is better than switch, because compiler is better in optimizing a switch-statement than an if-statement. The compiler doesn't know if the order of evaluating the if-statements is important to you, and can't perform any optimizations there. You could be calling methods in the if-statements, influencing variables. With the switch-statement it knows that all clauses can be evaluated at the same time and can put them in whatever order is most efficient.

Q 2)

```
main()
{
int i=5;
```

```
change();
int i=10;
printf("%d",i)
}
change()
{
?(need to write code here)
}
```

in above statement print the 5 with respect to the change function.

Ans. #define printf(a,b) print("5");

Description:

defined in above of main function is also It Prints 5.

Printf function is a Built in function Even it can be used as a User defined function.

Q 3)

```
#define printd(expr) printf(#expr “ %d”,expr)
main()
{
int a=10,b=2;
printd(a/b);
}
```

What is o/p?

Ans: a/b=5;

#expr treats that variable names.

Q.1 write a function which takes 7 as input and returns 4 and vice versa ?

```
A.1 int func1(int p) // input p should be 7 or 4
{
int i;
i=(p^3);
return i;
}
```

Q.2 what is stack flow after the statement being executed ?

OR

How these four things gets into picture 1. Compiler 2. C Language 3. Glibc 4. Kernel after the below statement?  
char \*p=((char \*)malloc (10));

A.2

1. C Language :- It specifies the header file <stdlib.h> ,which will check the prototype.

2. Compiler :- It takes the output of the preprocessor, and the source code, and generates assembler source code it is kept in an object file.

3. Glibc :- The glibc package contains standard libraries.In order to save disk space and memory, as well as to make upgrading easier, common system code is kept in one place and shared between programs.

This particular package contains the most important sets of shared libraries: the standard C library and the standard math library. Without these two libraries, a Linux system will not function

So object file generated now gets linked with the Glibc where in implementation of malloc is being defined.

There are other APIs like uclibc , bionic libc which can be linked with different OS (Linux, BCD, QNX) in .Config file while building kernel, depending on the requirement of the user.

3. Kernel :- Glibc calls the kernel via brk() system call.

1. Allocated 2D memory with malloc.

Ans: Pointers can be easily used to create a 2D array in C using malloc. The idea is to first create a one dimensional array of pointers, and then, for each array entry, create another one dimensional array. Here's a sample code:

```
double**theArray;
theArray=(double**)malloc(arraySizeX*sizeof(double*));
for(int i=0; i<arraySizeX;i++)
theArray[i]=(double*)malloc(arraySizeY*sizeof(double));
```

-----  
2.

```
int * fun()
{
static int i;
i=20;
return (&i);
}
```

Ans: its depend on OS implementation because static variable store in Data segment but it will try to return the value from stack so some times it works some time it does not work because it is dependent on how OS have property of memory distribution.

-----  
3.

```
int fun(int x)
{
int y;
y=x-1;
return((x&y)==0);
}
```

Ans:if value is 2 power n then it will return 1 else return 0 ;

sol: case 1: x=4 and then y=4-1=3

x=0100 and y =0011

So ((0100 & 0011)==0) it is true it will return 1.

Case2: x=6 and the y=5

x=0110 and y=0101

So ( (0110 & 0101)==0) it is not true so it will return 0.

conclusion: if value is 2 power n then it will return 1 else return 0 ;

---

Que- How can we pass entire array (all element of array) to the function.

Ans- for passing an array, wrap it in a structure:

for example-

```
struct xyz
{
int array[100];
};

int somefunc(struct xyz large)
{
...
}

void anotherfunc(void)
{
struct xyz a;
printf("%d\n", somefunc(a));
}
#####
#####
```

Que- What is the O/P of following program?

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

```
main()
{
char arr[]="string";

printf("%c",*arr);

arr++;

printf("%c",*arr);

}
```

Ans- Error !!!

error Line 9 : '++' needs l-value

A side effect of arr++ is an attempt to change the value of arr.we can't use statement like , 'arr++' because , arr is a const pointer to the allocated memory

```
#####
```

Q.1 predict the output for this code snippet with reason?

```
main()
{
 int (*ptr)[5];
```

```
char *p;
```

```
int arr[5]={1,2,3,4,5};
ptr=arr;
p=arr;
printf("ptr=%u \n p=%u",ptr,p);
p++;
ptr++;
printf("ptr=%u \n p=%u",ptr,p);
```

```
printf("*p=%u\n *ptr=%u",*p,*ptr);
```

```
printf("**ptr=%u\n",**ptr);
```

```
}
```

```
A.1 ptr=2402524240 p=2402524240
```

ptr=2402524260 p=2402524241 ( ptr is pointer to array so increments by 20 bytes int size x array size , while p is pointer to char so increments by 1 byte)

\*p=0 ( p is char pointer so it takes first byte of 4 bytes assigned that is zero here  
0000 0000 0000 0001 , value can be diff if it is big endian)

\*ptr=2402524260 ( single dereference cant give u value as it is pointer to array,we have to double dereference it for getting value)

\*\*ptr= 0; ( it gives value to next array memory is not assigned so it can take any value)

```
#####
```

Que- X % Y (X modulus Y )

Given- y is a power of 2.

calculate X % Y without using % operator.

Solution-

$$x \text{ modulo } y = (x \& (y - 1))$$

```
#####
Que- int a= some bit stream;
int b = another bit stream;
only 1 bit difference between 'a' and 'b'. find out the position of mismatch bit.
```

Ans-

```
#include <iostream>
using namespace std;
int main()
{
 int a=1123,b=1091; // declared a and b with integer and difference of their bits is only 1
 int c,d;
 int count=0;
 c= a ^ b ; //
 d= c-1;
 if((c & d) ==0)
 {
 while(c !=0)
 {
 count++;
 c=(c>>1);
 }
 }
 cout<<count; // return position where bits are mismatched
}
```

```
#####
Que-
```

I have a bit stream and without using for loop, I want to set m bit to n bit.

(m >n)

eg:

Bitstream = 10 10 10 10 01 00

m= 10

n=4,

O/P- 00 01 11 11 00 00

Solution :

$$(-1 << n) \& ((1 << m) - 1)$$

Note: given solution set all the bit within the range but turn off the bits which are beyond the range. to overcome this problem we can use this logic.

```
num | ((1<<(m-n+1))-1) << (n-1)
```

```
#####
#####
```

```
strtok
char *strtok(char *str, const char *delim);
```

The strtok() function parses a string into a sequence of tokens. On the first call to strtok() the string to be parsed should be specified in str. In each subsequent call that should parse the same string, str should be NULL. The delim argument specifies a set of bytes that delimit the tokens in the parsed string. The caller may specify different strings in delim in successive calls that parse the same string. Each call to strtok() returns a pointer to a null-terminated string containing the next token. The string does not include the delimiting byte. If no more tokens are found, strtok() returns NULL.

```
#include<string.h>

int main(){
char str[]="hello-this is rider";
char *pch;
char* a[10];
int i=0,j;
pch=strtok(str," "); //we can add more special character

while(pch!=NULL){
printf("%s\n",pch);
a[i]=pch;
i++;
pch=strtok(NULL," ");
}

reverseWords(a,i);
printf("Reverse:\n");

for(j=0;j<i;j++)
printf("%s\n", a[j]);
return 0;
}
```

O/P:  
rider

---

```
void main()
{
```

```

int a[2][3][2] = {{{2,4},{7,8},{3,4}},{ {2,2},{2,3},{3,4}}};
printf("0x%u 0x%u 0x%u %d \n",a,*a,**a,***a);
printf("0x%u 0x%u 0x%u %d \n",a+1,*a+1,**a+1,***a+1);
}

```

O/P:  
0x382857360 0x382857360 0x382857360 2  
0x382857384 0x382857368 0x382857364 3

**Explanation:**

The base address of array is 0x382857360.

As we know element at location [1][1][1] can be accessed as “a[1][1][1]” or \*( \*( \*(a + 1) + 1) + 1).

T \*p; // p is a pointer to an object of type T

When a pointer p is pointing to an object of type T, the expression \*p is of type T.

For example a is of type array of 2 two dimensional arrays.

The type of the expression \*a is “array of arrays (i.e. two dimensional array)”.

Based on the above concept translating the expression \*( \*( \*(buffer + 1) + 1) + 1).

For a three-dimensional array, reference to an array has type "pointer to array of 3 ints, of which each element as an array of 2 ints".

Therefore, a+1 is pointing to the memory location of first element of the second 2D array, a.

So,

a - An array of 2 two dimensional arrays, i.e. its type is “array of 2 two dimensional arrays”.

a + 1 - displacement for 2nd element in the array of 5 two dimensional arrays.

i.e a+1 is equivalent to a[1][0][0]

Hence 382857360 + 3\*(2 ints \* 4 bytes) = 382857384

Similarly, for \*a+1

\*(a + 0) + 1 - displacement to access 1th element in the array of 3 one dimensional arrays.

Therefore \*a+1 is equivalent to a[0][1][0]

Similarly, for \*\*a+1

\*( \*(a + 0) + 0) + 1 - displacement to get element at 1th position in the single dimension array of integers.

Therefore \*\*a+1 is equivalent to a[0][0][1]

Similarly, for \*\*\*a+1 is equivalent to value at a[0][0][1]

\*( \*( \*(a+ 0) + 0) + 0) - accessing the element at 0th position (the overall expression type is int now).

### String Literals

```

char *string = "English"; /* string literal */
char array[] = "English"; /* Char array not a string literal */
array[0] = 'V' ; /*Legal, no error*/
string[0] = 'V'; /* Undefined */

```

ANSI C standard defines String literals as " A string has type ``array of characters" and storage class static and is initialized with the given characters. Whether identical string literals are distinct is implementation defined, and the behavior of a program that attempts to alter a string literal is undefined." Trying to modify string literal will give segmentation fault on Linux however it may work on other OS or on processors which do not have memory protection.

In gcc/Linux string literal [char \*ptr] is stored read-only area hence you get a segmentation fault. But if non gcc compiler stores string literal in Read-Write area OR if the processor does not have MMU and OS does not offer memory protection program will work without any error. This is because C language standard does not clearly define what should be result of trying to modify string literals; C standard had left it to compilers/OS/Processor to use in best way they can.

Note that

### Structure alignment

1. Memory is always byte addressable.
2. For speed optimization compiler by default uses alignment which is same as data bus size. For ex. In case of machine 32-bit data bus alignment is done in multiples of 4-bytes.
3. In case of data types smaller than size of data bus padding bytes are added so that each variable is aligned.
4. Problem with padding: In real communication and in storage padding is never used as it is unnecessary wastage of time space. But if the padded structures used to reference protocol header or in media files data it could lead to incorrect results because of difference in alignment of actual data verses padded structures.
5. To avoid such issues compilers provide #pragma pack [see compiler document for specific syntax]. With this pragma programmer can instruct compiler not to add any padding and all the structures must be packed even though it means compromise of efficiency.

### Precedence & Associativity

```
#include<stdio.h>
main()
{
int a =10, b=20, c=0;
c=a+++b; /*No spaces in between*/
printf("a=%d, b=%d, c=%d\n",a,b,c);

return 0;
}
```

ANS:a=11, b=20 c=30

1. This is because ++ has higher precedence than +
2. ++ has "Right to Left" associativity hence it gets associated with a and becomes a++ i.e. post increment.

| Q . N o | Question                                                                                                                                              | Output                                                                                                                                                        | Discussion                                                                                                                                                                 |
|---------|-------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1       | <pre>char *const ptr; char ch1,ch2; ptr = &amp;ch1; ptr = &amp;ch2;</pre> <p>What is value stored in ptr ?</p>                                        | <p>Compilation Error:<br/>day1.c:6:1: error:<br/>assignment of read-only variable ‘ptr’<br/>day1.c:7:1: error:<br/>assignment of read-only variable ‘ptr’</p> | <p>char *const ptr; ==&gt;<br/>Address is constant ,<br/>value can be<br/>changed.<br/>Char const *ptr; ==&gt;<br/>Value is constant ,<br/>address can be<br/>changed.</p> |
| 2       | <pre>char arr[7] = "HELLO123"; printf("\n arr : %s",arr); printf("\n arr2 %c \n",*arr);</pre>                                                         | <p>arr : HELLO12<br/>arr2 : H</p>                                                                                                                             |                                                                                                                                                                            |
| 3       | Explain situation to use double pointer                                                                                                               | <pre>int *p; func(&amp;p);  void func(int **q){ *q = malloc(sizeof(int)); }</pre>                                                                             |                                                                                                                                                                            |
| 4       | <pre>What the value of p1 &amp; p2; char *str = "smartplay"; int p1 = sizeof(str); Int p2 = strlen(str);  printf("\n p1 : %d p2: %d \n",p1,p2);</pre> | <p>P1 = 6;<br/>P2 = 5 ;</p>                                                                                                                                   | <p>Sizeof operator returns the size of string including null character<br/><br/>Strlen function returns length of a string excluding null character.</p>                   |
| 5       | <pre>char *str1 = "smartplay"; char *str2[] = "smartplay";  printf("\n str1 : %u str2: %u \n",sizeof(str1),sizeof(st r2));</pre>                      | <p>Compilation error at *str2[].<br/>error: invalid initializer</p>                                                                                           | <p>str2 pointer should be memory allocated.<br/><br/>Char *str2 = (char *) malloc(size(5));<br/><br/>Do dynamic allocation to avoid such errors.</p>                       |

|   |                                                                                                                                                                                                                                                                        |       |                                                                   |
|---|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------|-------------------------------------------------------------------|
| 6 | <pre>#define TRUE 0 #define FALSE -1 #define NULL 0  void main(){  if(TRUE) printf("TRUE");  else if(FALSE) printf("FALSE");  Else printf("NULL");  }</pre>                                                                                                            | FALSE | -1 = FF Consider as TRUE                                          |
| 7 | <pre>enum actor { abc = 5, xyz = -2, gcd, hcf }; void main(){  enum actor a =0; switch(a) { case abc: printf("abc"); break; case xyz: printf("xyz"); break; case gcd: printf("gcd"); break; case hcf: printf("hcf"); break; default: printf("default"); break; }</pre> | hcf   | xyz = -2,<br>gcd, // gets -1 (xyz + 1)<br>hcf // gets 0 ( gcd +1) |

|    |                                                                                                                                                                                                        |                                                                                                                                                                 |                                                                                                                                                                                                                              |
|----|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 8  | <pre>switch(*(1 + "AB" "CD" + 1)) { case 'A': printf("A"); break; case 'B': printf("B"); break; case 'C': printf("C"); break; case 'D': printf("D"); break; default: printf("default"); break; }</pre> | C                                                                                                                                                               | <p>1 +ABCD + 1<br/>2 + ABCD<br/>A = 0 , B = 1 C = 2 , D = 3</p> <p>Expected value is 2 , So it reach case C:</p> <p>If switch((1 + "AB" "CD" + 1)) then its an compilation error : error: switch quantity not an integer</p> |
| 9  | <pre>struct student { int rno; char name[10]; }st;  void main(){  st.rno = 10; st.name = "hello";</pre>                                                                                                | Compilation Error :<br><br>error: incompatible types when assigning to type 'char[10]' from type 'char *'                                                       | <p>st.name = "hello"; is not allowed</p> <p>strcpy(st.name,"hello"); is allowed</p>                                                                                                                                          |
| 10 | <pre>int xx = NULL; int yy = 10; int zz = 0;  zz = xx + yy;  printf("Z : %d \n ",zz);</pre>                                                                                                            | Z = 10;                                                                                                                                                         | NULL Considered as a 0.                                                                                                                                                                                                      |
| 11 | <pre>printf("char : %c ",59); printf("int : %d ",59); printf("string : %s ",59);</pre>                                                                                                                 | Char - ;<br>Int - 59<br>String -<br>Segmentation fault                                                                                                          |                                                                                                                                                                                                                              |
| 12 | <pre>int i = 4,5,6; Int l = (4,5,6);</pre>                                                                                                                                                             | <p>First statement -<br/>Not allowed - error:<br/>expected identifier or '(' before numeric constant.</p> <p>Second statement -<br/>l = 6 will be assigned.</p> |                                                                                                                                                                                                                              |

|   |                     |        |  |
|---|---------------------|--------|--|
| 1 | int l = -1;         | l = 0; |  |
| 3 | while(+(+i) != 0) { |        |  |
|   | ++i;                |        |  |
|   | }                   |        |  |

.....

sorry for wrong answer in 1st question. please check it.

Q 1) if else statement is better than switch case if yes then why?

switch statement is better than if-else, because compiler is better in optimizing a switch-statement than an if-statement.

The compiler doesn't know if the order of evaluating the if-statements is important to you, and can't perform any optimizations there. You could be calling methods in the if-statements, influencing variables. With the switch-statement it knows that all clauses can be evaluated at the same time and can put them in whatever order is most efficient.

Q 1) if else statement is better than switch case if yes then why?

If else statement is better than switch, because compiler is better in optimizing a switch-statement than an if-statement.

The compiler doesn't know if the order of evaluating the if-statements is important to you, and can't perform any optimizations there. You could be calling methods in the if-statements, influencing variables. With the switch-statement it knows that all clauses can be evaluated at the same time and can put them in whatever order is most efficient.

Q 2)

```
main()
{
int i=5;
change();
int i=10;
printf("%d",i)
}
change()
{
?(need to write code here)
}
```

in above statement print the 5 with respect to the change function.

Ans. #define printf(a,b) print("5");

Description:

defined in above of main function is also It Prints 5.

Printf function is a Built in function Even it can be used as a User defined function.

Q 3)

```
#define printd(expr) printf(#expr " %d",expr)
main()
{
int a=10,b=2;
printd(a/b);
}
```

What is o/p?

Ans:  $a/b=5;$   
#expr treats that variable names.

Q.1 write a function which takes 7 as input and returns 4 and vice versa ?

```
A.1 int func1(int p) // input p should be 7 or 4
{
int i;
i=(p^3);
return i;
}
```

Q.2 what is stack flow after the statement being executed ?

OR

How these four things gets into picture 1. Compiler 2. C Language 3. Glibc 4. Kernel after the below statement?

```
char *p=((char *)malloc (10));
```

A.2

1. C Language :- It specifies the header file <stdlib.h> ,which will check the prototype.

2. Compiler :- It takes the output of the preprocessor, and the source code, and generates assembler source code it is kept in an object file.

3. Glibc :- The glibc package contains standard libraries.In order to save disk space and memory, as well as to make upgrading easier, common system code is kept in one place and shared between programs.

This particular package contains the most important sets of shared libraries: the standard C library and the standard math library. Without these two libraries, a Linux system will not function

So object file generated now gets linked with the Glibc where in implementation of malloc is being defined.

There are other APIs like uclibc , bionic libc which can be linked with different OS (Linux, BCD, QNX) in .Config file while building kernel, depending on the requirement of the user.

3. Kernel :- Glibc calls the kernel via brk() system call.

1.Allocated 2D memory with malloc.

Ans: Pointers can be easily used to create a 2D array in C using malloc. The idea is to first create a one dimensional array of pointers, and then, for each array entry, create another one dimensional array. Here's a sample code:

```
double**theArray;
theArray=(double**)malloc(arraySizeX*sizeof(double*));
for(int i=0; i<arraySizeX;i++)
theArray[i]=(double*)malloc(arraySizeY*sizeof(double));

2.
int * fun()
{
static int i;
i=20;
return (&i);
}
```

Ans: its depend on OS implementation because static variable store in Data segment but it will try to return the value from stack so some times it works some time it does not work because it is dependent on how OS have property of memory distribution.

```

3.
int fun(int x)
{
 int y;
 y=x-1;
 return((x&y)==0);
}
```

Ans: if value is 2 power n then it will return 1 else return 0 ;

sol: case 1: x=4 and then y=4-1=3

x=0100 and y =0011

So ((0100 & 0011)==0) it is true it will return 1.

Case2: x=6 and the y=5

x=0110 and y=0101

So ((0110 & 0101)==0) it is not true so it will return 0.

conclusion: if value is 2 power n then it will return 1 else return 0 ;

Que- How can we pass entire array (all element of array) to the function.

Ans- for passing an array, wrap it in a structure:

for example-

```
struct xyz
{
int array[100];
};

int somefunc(struct xyz large)
{
...
}
void anotherfunc(void)
{
struct xyz a;
printf("%d\n", somefunc(a));
}
#####
```

Que- What is the O/P of following program?

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

```
main()
{
char arr[]="string";

printf("%c",*arr);

arr++;

printf("%c",*arr);
```

}

```
Ans- Error !!!
error Line 9 : '++' needs l-value
```

A side effect of arr++ is an attempt to change the value of arr. we can't use statement like , 'arr++' because , arr is a constant pointer to the allocated memory

```
#####
#####
```

Q.1 predict the output for this code snippet with reason?

```
main()
{
int (*ptr)[5];

char *p;

int arr[5]={1,2,3,4,5};
ptr=arr;
p=arr;
printf("ptr=%u \n p=%u",ptr,p);
p++;
ptr++;
printf("ptr=%u \n p=%u",ptr,p);

printf("*p=%u\n *ptr=%u",*p,*ptr);

printf("**ptr=%u\n",**ptr);
```

```
}
```

A.1 ptr=2402524240 p=2402524240

ptr=2402524260 p=2402524241 ( ptr is pointer to array so increments by 20 bytes int size x array size , while p is pointer to char so increments by 1 byte)

\*p=0 ( p is char pointer so it takes first byte of 4 bytes assigned that is zero here  
0000 0000 0000 0001 , value can be diff if it is big endian)  
\*ptr=2402524260 ( single dereference cant give u value as it is pointer to array,we have to double dereference it for getting value)

\*\*ptr= 0; ( it gives value to next array memory is not assigned so it can take any value)

```
#####
#####
```

Que- X % Y (X modulus Y )

Given- y is a power of 2.

calculate X % Y without using % operator.

Solution-

```
x modulo y = (x & (y - 1))

#####
#####
```

Que- int a= some bit stream;  
int b = another bit stream;  
only 1 bit difference between 'a' and 'b'. find out the position of mismatch bit.

Ans-

```
#include <iostream>
using namespace std;
int main()
{
int a=1123,b=1091; // declared a and b with integer and difference of their bits is only 1
int c,d;
int count=0;
c= a ^ b ; //
d= c-1;
if((c & d) ==0)
{
while(c !=0)
{
count++;
c=(c>>1);
}
}
cout<<count; // return position where bits are mismatched
}
```

```
#####
#####
```

Que-  
I have a bit stream and without using for loop, I want to set m bit to n bit.  
(m >n)  
eg:  
Bitstream = 10 10 10 10 01 00  
m= 10  
n=4,  
O/P- 00 01 11 11 00 00

Solution :  
 $(-1 \ll n) \& ((1 \ll m) - 1)$

Note: given solution set all the bit within the range but turn off the bits which are beyond the range.  
to overcome this problem we can use this logic.

```
num | (((1<<(m- n+1))-1) << (n-1))
```

```
#####
#####
```

**strtok**  
**char \*strtok(char \*str, const char \*delim);**

The strtok() function parses a string into a sequence of tokens. On the first call to strtok() the string to be parsed should be specified in str. In each subsequent call that should parse the same string, str should be NULL. The delim argument specifies a set of bytes that delimit the tokens in the parsed string. The caller may specify different strings in delim in successive calls that parse the same string. Each call to strtok() returns a pointer to a null-terminated string containing the next token. This string does not include the delimiting byte. If no more tokens are found, strtok() returns NULL.

```
#include<string.h>

int main(){
char str[]="hello-this is rider";
char *pch;
char* a[10];
int i=0,j;
pch=strtok(str," "); //we can add more special character
```

```
while(pch!=NULL){
printf("%s\n",pch);
a[i]=pch;
i++;
pch=strtok(NULL," ");
}
```

```
reverseWords(a,i);
printf("Reverse:\n");
```

```
for(j=0;j<i;j++)
printf("%s\n", a[j]);
return 0;
}
```

O/P:  
rider  
is  
hello-this

---



---

```
void main()
{
int a[2][3][2] = {{{2,4},{7,8},{3,4}},{ {2,2},{2,3},{3,4}}};
printf("0x%u 0x%u 0x%u %d \n",a,*a,**a,***a);
printf("0x%u 0x%u 0x%u %d \n",a+1,*a+1,**a+1,***a+1);
}
```

O/P:  
0x382857360 0x382857360 0x382857360 2  
0x382857384 0x382857368 0x382857364 3

Explanation:

The base address of array is 0x382857360.

As we know element at location [1][1][1] can be accessed as "a[1][1][1]" or \*( \*( \*(a + 1) + 1) + 1).

T \*p; // p is a pointer to an object of type T  
When a pointer p is pointing to an object of type T, the expression \*p is of type T.  
For example a is of type array of 2 two dimensional arrays.  
The type of the expression \*a is "array of arrays (i.e. two dimensional array)".

Based on the above concept translating the expression \*( \*( \*(buffer + 1) + 1) + 1).

For a three-dimensional array, reference to an array has type "pointer to array of 3 ints, of which each element as an array of 2 ints".  
Therefore, a+1 is pointing to the memory location of first element of the second 2D array, a.

So,  
a - An array of 2 two dimensional arrays, i.e. its type is "array of 2 two dimensional arrays".  
a + 1 - displacement for 2nd element in the array of 5 two dimensional arrays.

i.e a+1 is equivalent to a[1][0][0]  
Hence 382857360 + 3\*(2 ints \* 4 bytes) = 382857384

Similarly, for \*a+1  
\*(a + 0) + 1 - displacement to access 1th element in the array of 3 one dimensional arrays.  
Therefore \*a+1 is equivalent to a[0][1][0]

Similarly, for \*\*a+1  
\*( \*(a + 0) + 0) + 1 - displacement to get element at 1th position in the single dimension array of integers.  
Therefore \*\*a+1 is equivalent to a[0][0][1]

Similarly, for \*\*\*a+1 is equivalent to value at a[0][0][1]  
\*( \*( \*(a + 0) + 0) + 0) - accessing the element at 0th position (the overall expression type is int now).

String Literals  
char \*string = "English"; /\* string literal \*/  
char array[] = "English"; /\* Char array not a string literal \*/  
array[0] = 'V' ; /\*Legal, no error\*/  
string[0] = 'V'; /\* Undefined \*/

ANSI C standard defines String literals as " A string has type ``array of characters'' and storage class static and is initialized with the given characters. Whether identical string literals are distinct is implementation defined, and the behavior of a program that attempts to alter a string literal is undefined." Trying to modify string literal will give segmentation fault on Linux however it may work on other OS or on processors which do not have memory protection.

In gcc/Linux string literal [char \*ptr] is stored read-only area hence you get a segmentation fault. But if non gcc compiler stores string literal in Read-Write area OR if the processor does not have MMU and OS does not offer memory protection this program will work without any error. This is because C language standard does not clearly define what should be result of code trying to modify string literals; C standard had left it to compilers/OS/Processor to use in best way they can.

Note that

Structure alignment

1. Memory is always byte addressable.
2. For speed optimization compiler by default uses alignment which is same as data bus size. For ex. In case of machine with 32-bit data bus alignment is done in multiples of 4-bytes.
3. In case of data types smaller than size of data bus padding bytes are added so that each variable

is aligned.

4. Problem with padding: In real communication and in storage padding is never used as it is unnecessary wastage of time & space. But if the padded structures used to reference protocol header or in media files data it could lead to incorrect results because of difference in alignment of actual data verses padded structures.

5. To avoid such issues compilers provide #pragma pack [see compiler document for specific syntax]. With this pragma programmer can instruct compiler not to add any padding and all the structures must be packed even though it mean compromise of efficiency.

```
Precedence & Associativity
#include<stdio.h>
main()
{
int a =10, b=20, c=0;
c=a+++b; /*No spaces in between*/
printf("a=%d, b=%d, c=%d\n",a,b,c);

return 0;
}
```

ANS:a=11, b=20 c=30

1. This is because ++ has higher precedence than +
2. ++ has "Right to Left" associativity hence it gets associated with a and becomes a++ i.e. post increment.

## Synchronization mechanisms inside Linux kernel

Link:

<https://kerneltweaks.wordpress.com/2015/03/20/quick-guide-for-choosing-correct-synchronization-mechanism-inside-linux-kernel/>

Hello folks, today I am going to talk about the synchronization mechanism which is available in Linux kernel. This post may help you to choose right synchronization mechanism for uni-processor or SMP system. Choosing wrong mechanism can cause crash to kernel or it can damage any hardware component. Before we begin, let's closely examine three terminology which will be frequently used in this post.

1. Critical Region: A critical section is a piece of code which should be executed under mutual exclusion. Suppose that, two threads are updating the same variable which is in parent process's address space. So the code area where both thread access/update the shared variable/resource is called as a Critical Region. It is essential to protect critical region to avoid collision in code/system.

2. Race Condition: So developer has to protect Critical Region such that, at one time instance there is only one thread/process which is passing under that region( accessing shared resources). If Critical Region doesn't protected with the proper mechanism, then there are chances of Race Condition.

Finally, a *race condition* is a flaw that occurs when the timing or ordering of events affects a program's correctness. by using appropriate synchronization mechanism or properly protecting Critical Region we can avoid/reduce the chance of this flaw.

3. Deadlock: This is the other flaw which can be generated by NOT using proper synchronization mechanism. It is a situation in which two thread/process sharing the same resource are effectively preventing each other from accessing the resource, resulting in both programs ceasing to function.

So the question comes to mind is "what is synchronization mechanism ?" Synchronization mechanism is set of APIs & objects which can be used to protect critical region and avoid deadlock/race condition.

Linux kernel provide couple of synchronization mechanism.

1. **Atomic operation:** This is the very simple approach to avoid race condition or deadlock. Atomic operators are operations, like add and subtract, which perform in one clock cycle (uninterruptible operation). The atomic integer methods operate on a special data type, `atomic_t`. **A common use of the atomic integer operations is to implement counters which is updated by multiple threads.** The kernel provides two sets of interfaces for atomic operations, one that operates on integers and another that operates on individual bits. All atomic functions are inline functions.

1. APIs for operations on integers:

```
atomic_t i /* defining atomic variable i */
atomic_set(&i, 10); /* atomically assign value(here 10) to atomic variable i */
atomic_inc(&i); /* atomically increment value of i variable (i++ atomically), so i = 11 */
atomic_dec(&i); /* atomically decrement value of i variable (i- atomically), so i = 10 */
atomic_add(4, &i); /* atomically add 4 with value of i (i = 10+4) */
atomic_read(&i); /* atomically read and return value of i (14) */
```

2. APIs for operates on individual bits:

Bit-wise APIs operate on any generic memory addresses. So there is no need to for explicitly defining an object with the type of `atomic_t`.

```
unsigned int i = 0; /* defining a normal variable (i = 0Xoooooooo)*/
set_bit(5, &i); /*atomically Set 5th bit of the variable i (i = 0X00000010) */
clear_bit(5, &i); /* atomically clear 5th bit of the variable i (i = 0Xoooooooo) */
```

2. **Semaphore**: This is another kind of synchronization mechanism which will be provided by the Linux kernel. When some process is trying to access semaphore which is not available, semaphore puts process on wait queue(FIFO) and puts task on sleep. That's why semaphore is known as a sleeping lock. After this processor is free to jump to other task which is not requiring this semaphore. As soon as semaphore get available, one of task from wait queue in invoked.

There two flavors of semaphore is present.

- Basic semaphore
- Reader-Writer Semaphore

When a multiple threads/process wants to share data, in the case where read operation on data is more frequent and write operation is rare. In this scenario Reader-Writer Semaphore is used. Multiple thread can read a data by same time. The data will be only locked(all other read thread should wait) when one thread write/update data. On the other side writers has to wait until all the readers release the read lock. When writer process release lock the reader from wait-queue(FIFO) will get invoked.

Couple of observations about nature of semaphore :

1. Semaphore puts a task on sleep. So the semaphore can be only used in process context. Interrupt context can not sleep.
2. Operation to put task on sleep is time consuming(overhead) for CPU. So semaphore is suitable for lock which is holding for long term. Sleeping and invoking task over kills CPU if semaphore is locked and unlocked for short time via multiple tasks.
3. A code holding a semaphore can be preempted. It does not disable kernel preemption.
4. After disabling interrupts from some task, semaphore should not acquired. Because task would sleep if it fails to acquire the semaphore, at this time the interrupt has been disabled and current task cannot be scheduled out.

5. Semaphore wait list is FIFO in nature. So the task which tried to acquire semaphore first will be waken up from wait list first.
  6. Semaphore can be acquired/release from any process/thread.
3. **Spin-lock:** This is special type of synchronization mechanism which is preferable to use in multi-processor(SMP) system. Basically its a busy-wait locking mechanism until the lock is available. In case of unavailability of lock, it keeps thread in light loop and keep checking the availability of lock. Spin-lock is not recommended to use in single processor system. If some process\_1 has acquired a lock and other process\_2 is trying to acquire lock, in this case process 2 will spins around and keep processor core busy until it acquires lock. process\_2 will create a deadlock, it doesn't allow any other process to execute because CPU core is busy in light loop by semaphore.

Couple of observations about nature of spinlocks:

1. Spinlocks are very much suitable to use in interrupt(atomic) context because it doesn't put process/thread in sleep.
2. In the uni processor environment, if the kernel acquires a spin lock, it would disable preemption first ; if the kernel releases the spin lock, it would enable preemption. This is to avoid dead lock on uni processor system. EG: In uni processor system, thread\_1 has acquired spinlock. After that kernel preemption takes place, which puts thread\_1 to the stack and thread\_2 comes on CPU. Thread\_2 tries to acquire same spin-lock but which is not available. In this scenario, thread\_2 will keep CPU busy in light loop. This situation does not allow other thread to execute on CPU. This creates deadlock.
3. Spin-locks are not recursive
4. Special care must be taken in case where spin-lock is shared b/w interrupt handler and thread. Local interrupts must be disabled on the same CPU(core) before acquiring spin-lock. In the case where interrupt occurs on a different processor, and it spins on the same lock, does not cause deadlock because the processor who acquires lock will be able to release the lock using the other core. EG: Suppose that an interrupt handler to interrupt kernel code while the lock is acquired by thread. The interrupt handler spins, wait for the lock to become available. The locker thread, does not run until the interrupt handler completes. This can cause dead lock.
5. When data is shared between two tasklets, there is no need to disable interrupts because tasklet does not allow another running tasklet on the same processor. [Here](#) you can get more details about nature of tasklets.

There are two flavors of spin-lock present.

- Basic spin-lock
- Reader-Writer Spin-lock

With increasing the level of concurrency in Linux kernel read-write variant of spin-lock is introduced. This lock is used in the scenario where many readers and few writers are present. Read-write spin-lock can have multiple readers at a time but only one writer and there can be no readers while there is a writer. Any reader will not get lock until writer finishes it.

4. **Sequence Lock:** This is very useful synchronization mechanism to provide a lightweight and scalable lock for the scenario where many readers and a few writers are present. Sequence lock maintains a counter for sequence. When the shared data is written, a lock is obtained and a sequence counter is incremented by 1. Write operation makes the sequence counter value to odd and releasing it makes even. In case of reading, sequence counter is read before and after reading the data. If the values are the same which indicates that a write did not begin in the middle of the read. In addition to that, if the values are even, a write operation is not going on. Sequence lock gives the high priority to writers compared to readers. An acquisition of the write lock always succeeds if there are no other writers present. Pending writers continually cause the read loop to repeat, until there are no longer any writers holding the lock. So reader may sometimes be forced to read the same data several times until it gets a valid copy(writer releases lock). On the other side writer never waits until and unless another writer is active.

So, every synchronization mechanism has its own pros and cons. Kernel developer has to smartly choose appropriate synchronization mechanism based on pros and cons.

Stay tuned !!!!

- 1.
- 2.
- 3.
- 4.
- 5.
6. Race condition

A race condition is an undesirable situation that occurs when a device or system attempts to perform two or more operations at the same time, but because of the nature of the device or system, the operations must be done in the proper sequence to be done correctly.

What is a race condition?

It occurs when the output and/or result of the process is critically dependent on the sequence or timing of other events i.e. e.g. 2 signals are racing to change the output first.

How do you detect them?

It leads to error which is difficult to localize.

How do you handle them?

Use Semaphores

And finally,

How do you prevent them from occurring?

One way to avoid race condition is using locking mechanism for resources. but locking resources can lead to deadlocks. which has to be dealt with.

7. Atomic Operation.

### **Definition - What does *Atomic Operation* mean?**

Atomic operations in concurrent programming are program operations that run completely independently of any other processes.

Atomic operations are used in many modern operating systems and parallel processing systems.

**Techopedia explains *Atomic Operation***

Atomic operations are often used in the kernel, the primary component of most operating systems. However, most computer hardware, compilers and libraries also provide varying levels of atomic operations.

In loading and storing, computer hardware carries out writing and reading to a word-sized memory. To fetch, add or subtract, value augmentation takes place through atomic operations. During an atomic operation, a processor can read and write a location during the same data transmission. In this way, another input/output mechanism or processor cannot perform memory reading or writing tasks until the atomic operation has finished.

Where data is being used by an atomic operation that is also in use by other atomic or non-atomic operations, it can only exist in either sequential processing environments or locking mechanisms have to be used to avoid data errors. Compare and swap is another method but does not guarantee data integrity for atomic operation results.

The problem comes when two operations running in parallel (concurrent operations) utilise the same data and a disparity between the results of the operations occurs. Locking locks variable data and forces sequential operation of atomic processes that utilize the same data or affect it in some way.

---

### 7.11. Avoid Race Conditions

A “race condition” can be defined as “Anomalous behavior due to unexpected critical dependence on the relative timing of events” [FOLDOC]. Race conditions generally involve one or more processes accessing a shared resource (such a file or variable), where this multiple access has not been properly controlled.

In general, processes do not execute atomically; another process may interrupt it between essentially any two instructions. If a secure program’s process is not prepared for these interruptions, another process may be able to interfere with the secure program’s process. Any pair of operations in a secure program must still work correctly if arbitrary amounts of another process’s code is executed between them.

Race condition problems can be notionally divided into two categories:

- Interference caused by untrusted processes. Some security taxonomies call this problem a “sequence” or “non-atomic” condition. These are conditions caused by processes running other, different programs, which “slip in” other actions between steps of the secure program. These other programs might be invoked by an attacker specifically to cause the problem. This book will call these sequencing problems.
- Interference caused by trusted processes (from the secure program’s point of view). Some taxonomies call these deadlock, livelock, or locking failure conditions. These are conditions caused by processes running the “same” program. Since these different processes may have the “same” privileges, if not properly controlled they may be able to interfere with each other in a way other programs can’t. Sometimes this kind of interference can be exploited. This book will call these locking problems.

8. Semaphore---> Binary semaphore, Counting Semaphore---

9. Pre-emptable and Non-Pre-emptable tasks

10. Critical region or critical Section.

11. Semaphore Vs Mutex

## Mutex vs Semaphore

What are the differences between Mutex vs Semaphore? When to use mutex and when to use semaphore?

Concrete understanding of Operating System concepts is required to design/develop smart applications. Our objective is to educate the reader on these concepts and learn from other expert geeks.

As per operating system terminology, mutex and semaphore are kernel resources that provide synchronization services (also called as *synchronization primitives*). *Why do we need such synchronization primitives? Won't be only one sufficient?* To answer these questions, we need to understand few keywords. Please read the posts on [atomicity](#) and [critical section](#). We will illustrate with examples to understand these concepts well, rather than following usual OS textual description.

### The producer-consumer problem:

*Note that the content is generalized explanation. Practical details vary with implementation.*

Consider the standard producer-consumer problem. Assume, we have a buffer of 4096 byte length. A producer thread collects the data and writes it to the buffer. A consumer thread processes the collected data from the buffer. Objective is, both the threads should not run at the same time.

### Using Mutex:

A mutex provides mutual exclusion, either producer or consumer can have the key (mutex) and proceed with their work. As long as the buffer is filled by producer, the consumer needs to wait, and vice versa.

At any point of time, only one thread can work with the *entire* buffer. The concept can be generalized using semaphore.

### Using Semaphore:

A semaphore is a generalized mutex. In lieu of single buffer, we can split the 4 KB buffer into four 1 KB buffers (identical resources). A semaphore can be associated with these four buffers. The consumer and producer can work on different buffers at the same time.

### Misconception:

There is an ambiguity between *binary semaphore* and *mutex*. We might have come across that a mutex is binary semaphore. *But they are not!* The purpose of mutex and semaphore are different. May be, due to similarity in their implementation a mutex would be referred as binary semaphore.

Strictly speaking, a mutex is **locking mechanism** used to synchronize access to a resource. Only one task (can be a thread or process based on OS abstraction) can acquire the mutex. It means there is ownership associated with mutex, and only the owner can release the lock (mutex).

Semaphore is **signaling mechanism** ("I am done, you can carry on" kind of signal). For example, if you are listening songs (assume it as one task) on your mobile and at the same time your friend calls you, an interrupt is triggered upon which an interrupt service routine (ISR) signals the call processing task to wakeup.

### General Questions:

1. *Can a thread acquire more than one lock (Mutex)?*

Yes, it is possible that a thread is in need of more than one resource, hence the locks. If any lock is not available the thread will wait (block) on the lock.

### 2. Can a mutex be locked more than once?

A mutex is a lock. Only one state (locked/unlocked) is associated with it. However, a *recursive mutex* can be locked more than once (POSIX complaint systems), in which a count is associated with it, yet retains only one state (locked/unlocked). The programmer must unlock the mutex as many number times as it was locked.

### 3. What happens if a non-recursive mutex is locked more than once?

Deadlock. If a thread which had already locked a mutex, tries to lock the mutex again, it will enter into the waiting list of that mutex, which results in deadlock. It is because no other thread can unlock the mutex. An operating system implementer can exercise care in identifying the owner of mutex and return if it is already locked by same thread to prevent deadlocks.

### 4. Are binary semaphore and mutex same?

No. We suggest to treat them separately, as it is explained signalling vs locking mechanisms. But a binary semaphore may experience the same critical issues (e.g. priority inversion) associated with mutex. We will cover these in later article.

A programmer can prefer mutex rather than creating a semaphore with count 1.

### 5. What is a mutex and critical section?

Some operating systems use the same word *critical section* in the API. Usually a mutex is costly operation due to protection protocols associated with it. At last, the objective of mutex is atomic access. There are other ways to achieve atomic access like disabling interrupts which can be much faster but ruins responsiveness. The alternate API makes use of disabling interrupts.

### 6. What are events?

The semantics of mutex, semaphore, event, critical section, etc... are same. All are synchronization primitives. Based on their cost in using them they are different. We should consult the OS documentation for exact details.

### 7. Can we acquire mutex/semaphore in an Interrupt Service Routine?

An ISR will run asynchronously in the context of current running thread. It is **not recommended** to query (blocking call) the availability of synchronization primitives in an ISR. The ISR are meant be short, the call to mutex/semaphore may block the current running thread. However, an ISR can signal a semaphore or unlock a mutex.

### 8. What we mean by “thread blocking on mutex/semaphore” when they are not available?

Every synchronization primitive has a waiting list associated with it. When the resource is not available, the requesting thread will be moved from the running list of processor to the waiting list of the synchronization primitive. When the resource is available, the higher priority thread on the waiting list gets the resource (more precisely, it depends on the scheduling policies).

### 9. Is it necessary that a thread must block always when resource is not available?

Not necessary. If the design is sure ‘what has to be done when resource is not available’, the thread can take up that work (a different code branch). To support application requirements the OS provides non-blocking API.

For example POSIX pthread\_mutex\_trylock() API. When mutex is not available the function returns immediately whereas the API pthread\_mutex\_lock() blocks the thread till resource is available.

## References:

<http://www.netrino.com/node/202>

<http://doc.trolltech.com/4.7/qsemaphore.html>

Also compare mutex/semaphores with Peterson’s algorithm and Dekker’s algorithm. A good reference is the *Art of Concurrency* book. Also explore reader locks and writer locks in Qt documentation.

## Exercise:

Implement a program that prints a message “An instance is running” when executed more than once in the same session. For example, if we observe word application or Adobe reader in Windows, we can see only one instance in the task manager. How to implement it?

Article compiled by **Venki**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner   Company Wise Coding Practice[Articles](#)[Operating Systems](#)[Process Synchronization](#)**About Venki**

Software Engineer

[View all posts by Venki →](#)**Recommended Posts:**

- [Peterson's Algorithm for Mutual Exclusion | Set 1 \(Basic C implementation\)](#)
- [CriticalSection](#)
- [Priority Inversion : What the heck !](#)
- [Inter Process Communication](#)
- [Deadlock Detection And Recovery](#)

---

**Difference between binary semaphore and mutex**

hey are NOT the same thing. They are used for different purposes!

While both types of semaphores have a full/empty state and use the same API, their usage is very different.

**Mutual Exclusion Semaphores**

Mutual Exclusion semaphores are used to protect shared resources (data structure, file, etc..). A Mutex semaphore is "owned" by the task that takes it. If Task B attempts to semGive a mutex currently held by Task A, Task B's call will return an error and fail.

Mutexes always use the following sequence:

- SemTake
- Critical Section
- SemGive

Here is a simple example:

Thread A

Thread B

Take Mutex

```

access data
...
Take Mutex <== Will block
...
Give Mutex access data <== Unblock
...
Give Mutex

```

**Binary Semaphore**

Binary Semaphore address a totally different question:

- Task B is pended waiting for something to happen (a sensor being tripped for example).
- Sensor Trips and an Interrupt Service Routine runs. It needs to notify a task of the trip.
- Task B should run and take appropriate actions for the sensor trip. Then go back to waiting.

|                         |                                          |
|-------------------------|------------------------------------------|
| Task A                  | Task B                                   |
| ...                     | Take BinSemaphore <== wait for something |
| Do Something Noteworthy | do something <== unblocks                |
| Give BinSemaphore       |                                          |

Note that with a binary semaphore, it is OK for B to take the semaphore and A to give it.

Again, a binary semaphore is NOT protecting a resource from access. The act of Giving and Taking a semaphore are fundamentally decoupled.

It typically makes little sense for the same task to do a give and a take on the same binary semaphore.

Mutex can be released only by thread that had acquired it, while you can signal semaphore from any other thread (or process), so semaphores are more suitable for some synchronization problems like producer-consumer.

On Windows, binary semaphores are more like event objects than mutexes.

[The Toilet example](#) is an enjoyable analogy:

315down  
Mutex:

vote Is a key to a toilet. One person can have the key - occupy the toilet - at the time. When finished, the person gives (frees) the key to the next person in the queue.

Officially: "Mutexes are typically used to serialise access to a section of re-entrant code that cannot be executed concurrently by more than one thread. A mutex object only allows one thread into a controlled section, forcing other threads which attempt to gain access to that section to wait until the first thread has exited from that section." Ref: Symbian Developer Library

(A mutex is really a semaphore with value 1.)

Semaphore:

Is the number of free identical toilet keys. Example, say we have four toilets with identical locks and keys. The semaphore count - the count of keys - is set to 4 at beginning (all four toilets are free), then the count value is decremented as people are coming in. If all toilets are full, ie. there are no free keys left, the semaphore count is 0. Now, when eq. one person leaves the toilet, semaphore is increased to 1 (one free key), and given to the next person in the queue.

Officially: "A semaphore restricts the number of simultaneous users of a shared resource up to a maximum number. Threads can request access to the resource (decrementing the semaphore), and can signal that they have finished using the resource (incrementing the semaphore)." Ref: Symbian Developer Library

share

answered Dec 6 '08 at 19:53



dlinsin

9,076113347

[22](#) Funny and simple example. – [Tony](#) Apr 21 '09 at 6:55

[177](#) ... but this is regarding mutex vs counting semaphore. The question was asked about binary. – [Roman Nikitchenko](#) Nov 10 '09 at 7:47

[15](#) While what is said by david is correct, but it is NOT the answer to the question asked. Mladen Jankovic answer the answer to the question asked, where point is made to differentiate "binary-semaphore" vs "mutex". – [Ajeet Ganga](#) Aug 21 '11 at 15:45

[9](#) Unfortunately, this incorrect answer has more votes than the best answer by @Benoit – [NeonGlow](#) Apr 29 '13 at 9:18

[3](#) This answer is misleading. Should have compared only with Binary Semaphore. – [Hemanth](#) Mar 15 '14 at 8:27

[show 4 more comments](#)

Nice articles on the topic:

- [MUTEX VS. SEMAPHORES – PART 1: SEMAPHORES](#)
- [MUTEX VS. SEMAPHORES – PART 2: THE MUTEX](#)
- [MUTEX VS. SEMAPHORES – PART 3 \(FINAL PART\): MUTUAL EXCLUSION PROBLEMS](#)

#### From part 2:

The mutex is similar to the principles of the binary semaphore with one significant difference: the principle of ownership. Ownership is the simple concept that when a task locks (acquires) a mutex only it can unlock (release) it. If a task tries to unlock a mutex it hasn't locked (thus doesn't own) then an error condition is encountered and, most importantly, the mutex is not unlocked. If the mutual exclusion object doesn't have ownership then, irrelevant of what it is called, it is not a mutex.

A semaphore can be a Mutex but a Mutex can never be semaphore. This simply means that a binary semaphore can be used as Mutex, but a Mutex can never exhibit the functionality of semaphore. 2. Both semaphores and Mutex (at least the on latest kernel) are nonrecursive in nature. 3. No one owns semaphores, whereas Mutex are owned and the owner is held

responsible for them. This is an important distinction from a debugging perspective. 4. In case of Mutex, the thread that owns the Mutex is responsible for freeing it. However, in the case of semaphores, this condition is not required. Any other thread can signal to free the semaphore by using the `s m p s` (function. `e_ot`) 5. A Mutex, by definition, is used to serialize access to a section of reentrant code that cannot be executed concurrently by more than one thread. A semaphore, by definition, restricts the number of simultaneous users of a shared resource up to a maximum number 6. Another difference that would matter to developers is that semaphores are systemwide and remain in the form of files on the filesystem, unless otherwise cleaned up. Mutex are processwide and get cleaned up automatically when a process exits. 7. The nature of semaphores makes it possible to use them in synchronizing related and unrelated processes, as well as between threads. Mutex can be used only in synchronizing between threads and at most between related processes (the pthread implementation of the latest kernel comes with a feature that allows Mutex to be used between related processes). 8. According to the kernel documentation, Mutex are lighter when compared to semaphores. What this means is that a program with semaphore usage has a higher memory footprint when compared to a program having Mutex. 9. From a usage perspective, Mutex has simpler semantics when compared to semaphores.

Myth:

Couple of articles say that "binary semaphore and mutex are same" or "Semaphore with value 1 is mutex" but the basic difference is Mutex can be released only by the thread that had acquired it, while you can signal semaphore from any other thread

Key Points:

- A thread can acquire more than one lock (Mutex).
- A mutex can be locked more than once only if it's a recursive mutex, here lock and unlock for mutex should be same
- If a thread which had already locked a mutex, tries to lock the mutex again, it will enter into the waiting list of that mutex, which results in deadlock.
- Binary semaphore and mutex are similar but not same.
- Mutex is costly operation due to protection protocols associated with it.
- Main aim of mutex is to achieve atomic access or lock on resource

Modified question is - What's the difference between a mutex and a "binary" semaphore in "Linux"?

6 down  
vote

Ans: Following are the differences – i) Scope – The scope of mutex is within a process address space which has created it and is used for synchronization of threads. Whereas semaphore can be used across process space and hence it can be used for interprocess synchronization.

- ii) Mutex is lightweight and faster than semaphore. Futex is even faster.
- iii) Mutex can be acquired by same thread successfully multiple times with condition that it should release it same number of times. Other thread trying to acquire will block. Whereas in case of semaphore if same process tries to acquire it again it blocks as it can be acquired only once.

<http://www.geeksforgeeks.org/archives/9102> discusses in details.

5down  
vote      Mutex is locking mechanism used to synchronize access to a resource. Semaphore is signaling mechanism.

Its up to programmer if he/she wants to use binary semaphore in place of mutex.

A mutex is essentially the same thing as a binary semaphore and sometimes uses the same basic implementation. The differences between them are in how they are used. While a binary semaphore may be used as a mutex, a mutex is a more specific use-case, which allows extra guarantees:

1. Mutexes have a concept of an owner. Only the process that locked the mutex is supposed to unlock it. If the owner is stored by the mutex this can be verified at runtime.
2. Mutexes may provide priority inversion safety. If the mutex knows its current owner, it is possible to promote the priority of the owner whenever a higher-priority task starts waiting on the mutex.
3. Mutexes may also provide deletion safety, where the process holding the mutex cannot be accidentally deleted.

## Pure Functions

A function is called **pure function** if it always returns the same result for same argument values and it has no side effects like modifying an argument (or global variable) or outputting something. The only result of calling a pure function is the return value. Examples of pure functions are strlen(), pow(), sqrt() etc. Examples of impure functions are printf(), rand(), time(), etc.

If a function is known as pure to compiler then **Loop optimization** and **Subexpression elimination** can be applied to it. In GCC, we can mark functions as pure using the “pure” attribute.

```
__attribute__((pure)) return-type fun-name(arguments1, ...)
{
 /* Function body */
}
```

Following is an example pure function that returns square of a passed integer.

```
__attribute__((pure)) int my_square(int val)
{
```

```

 return val*val;
}

```

Run on IDE

Consider the below example

```

for (len = 0; len < strlen(str); ++len)
 printf("%c", toupper(str[len]));

```

Run on IDE

If “strlen()” function is not marked as pure function then compiler will invoke the “strlen()” function with each iteration of the loop, and if function is marked as pure function then compiler knows that value of “strlen()” function will be same for each call, that's why compiler optimizes the for loop and generates code like following.

```
int len = strlen(str);
```

```

for (i = 0; i < len; ++i)
 printf("%c", toupper((str[i])));

```

Run on IDE

Let us write our own pure function to calculate string length.

```

__attribute__ ((pure)) size_t my_strlen(const char *str)
{
 const char *ptr = str;
 while (*ptr)
 ++ptr;

 return (ptr - str);
}

```

Run on IDE

Marking function as pure says that the hypothetical function “my\_strlen()” is safe to call fewer times than the program says.

This article is compiled by “Narendra Kangralkar” and reviewed by GeeksforGeeks team. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## Exception handling and object destruction | Set 1

Predict the output of following C++ program.

```
#include <iostream>
using namespace std;
```

```

class Test {
public:
 Test() { cout << "Constructing an object of Test " << endl; }
 ~Test() { cout << "Destructing an object of Test " << endl; }
};

int main() {
 try {
 Test t1;
 throw 10;
 } catch(int i) {
 cout << "Caught " << i << endl;
 }
}

```

Run on IDE

Output:

```

Constructing an object of Test
Destructing an object of Test
Caught 10

```

When an exception is thrown, destructors of the objects (whose scope ends with the try block) is automatically called before the catch block gets executed. That is why the above program prints “Destructing an object of Test” before “Caught 10”.

What happens when an exception is thrown from a constructor? Consider the following program.

```

#include <iostream>
using namespace std;

class Test1 {
public:
 Test1() { cout << "Constructing an Object of Test1" << endl; }
 ~Test1() { cout << "Destructing an Object of Test1" << endl; }
};

```

```
class Test2 {
public:
 // Following constructor throws an integer exception
 Test2() { cout << "Constructing an Object of Test2" << endl;
 throw 20; }

 ~Test2() { cout << "Destructing an Object of Test2" << endl; }
};

int main() {
 try {
 Test1 t1; // Constructed and destructed
 Test2 t2; // Partially constructed
 Test1 t3; // t3 is not constructed as this statement never gets executed
 } catch(int i) {
 cout << "Caught " << i << endl;
 }
 }
}
```

Run on IDE

Output:

```
Constructing an Object of Test1
Constructing an Object of Test2
Destructing an Object of Test1
Caught 20
```

Destructors are only called for the completely constructed objects. When constructor of an object throws an exception, destructor for that object is not called.

As an exercise, predict the output of following program.

```
#include <iostream>
using namespace std;
```

```
class Test {
 static int count;
```

```
int id;

public:

 Test() {
 count++;
 id = count;
 cout << "Constructing object number " << id << endl;
 if(id == 4)
 throw 4;
 }

 ~Test() { cout << "Destructing object number " << id << endl; }

};

int Test::count = 0;

int main() {
 try {
 Test array[5];
 } catch(int i) {
 cout << "Caught " << i << endl;
 }
}
```

Run on IDE

We will be covering more of this topic in a separate post. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

[GATE CS Corner](#) [Company Wise Coding Practice](#)

[C](#)

[C++](#)

## Recommended Posts:

- [Templates and Static variables in C++](#)
- [Catching base and derived classes as exceptions](#)

- [Stack Unwinding in C++](#)
- [Exception Handling in C++](#)
- [Template Metaprogramming in C++](#)

## How to deallocate memory without using free() in C?

**Question:** How to deallocate dynamically allocated memory without using “free()” function.

**Solution:** Standard library function [realloc\(\)](#) can be used to deallocate previously allocated memory. Below is function declaration of “realloc()” from “stdlib.h”

```
void *realloc(void *ptr, size_t size);
```

Run on IDE

If “size” is zero, then call to realloc is equivalent to “free(ptr)”. And if “ptr” is NULL and size is non-zero then call to realloc is equivalent to “malloc(size)”.

Let us check with simple example.

```
/* code with memory leak */

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

int main(void)
{
 int *ptr = (int*)malloc(10);

 return 0;
}
```

Run on IDE

Check the leak summary with valgrind tool. It shows memory leak of 10 bytes, which is highlighted in red colour.

```
[narendra@ubuntu]$ valgrind --leak-check=full ./free
==1238== LEAK SUMMARY:
==1238== definitely lost: 10 bytes in 1 blocks.
==1238== possibly lost: 0 bytes in 0 blocks.
==1238== still reachable: 0 bytes in 0 blocks.
==1238== suppressed: 0 bytes in 0 blocks.

[narendra@ubuntu]$
```

Let us modify the above code.

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

```
#include <stdlib.h>

int main(void)
{
 int *ptr = (int*) malloc(10);

 /* we are calling realloc with size = 0 */
 realloc(ptr, 0);

 return 0;
}
```

Run on IDE

Check the valgrind's output. It shows no memory leaks are possible, highlighted in red color.

```
[narendra@ubuntu]$ valgrind --leak-check=full ./a.out
==1435== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 11 from 1)
==1435== malloc/free: in use at exit: 0 bytes in 0 blocks.
==1435== malloc/free: 1 allocs, 1 frees, 10 bytes allocated.
==1435== For counts of detected errors, rerun with: -v
==1435== All heap blocks were freed - no leaks are possible.
[narendra@ubuntu]$
```

This article is compiled by “Narendra Kangalkar” and reviewed by GeeksforGeeks team. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## 12. Spinning

## 13. Dead Lock

Spinning Vs Interrupt.

---

---

## Reentrant Function

A function is said to be reentrant if there is a provision to interrupt the function in the course of execution, service the interrupt service routine and then resume the earlier going on function, without hampering its earlier course of action. Reentrant functions are used in applications like hardware interrupt handling, recursion, etc.

The function has to satisfy certain conditions to be called as reentrant:

1. It may not use global and static data. Though there are no restrictions, but it is generally not advised. because the interrupt may change certain global values and resuming the course of action of the reentrant function with the new data may give undesired results.

2. It should not modify its own code. This is important because the course of action of the function should remain the same throughout the code. But, this may be allowed in case the interrupt routine uses a local copy of the reentrant function every time it uses different values or before and after the interrupt.

3. Should not call another non-reentrant function.

### Thread safety and Reentrant functions

Reentrancy is distinct from, but closely related to, thread-safety. A function can be thread-safe and still not reentrant. For example, a function could be wrapped all around with a mutex (which avoids problems in multithreading environments), but if that function is used in an interrupt service routine, it could starve waiting for the first execution to release the mutex. The key for avoiding confusion is that reentrant refers to only one thread executing. It is a concept from the time when no multitasking operating systems existed. (Source : [https://en.wikipedia.org/wiki/Reentrancy\\_\(computing\)](https://en.wikipedia.org/wiki/Reentrancy_(computing)))

Example of Non-Reentrant Functions:

```
// A non-reentrant example [The function depends on
// global variable i]

int i;

// Both fun1() and fun2() are not reentrant

// fun1() is NOT reentrant because it uses global variable i
int fun1()
{
 return i * 5;
}
```

```
// fun2() is NOT reentrant because it calls a non-reentrant
// function

int fun2()

{
 return fun1() * 5;
}
```

Run on IDE

**Example of Reentrant Functions:**

In the below code, fun2 is a reentrant function. If an interrupt that pauses its execution and shifts the control to fun1. After fun1 completes, the control is again transferred to fun2 and it reenters the execution phase.

// Both fun1() and fun2() are reentrant

```
int fun1(int i)
```

```
{
```

```
 return i * 5;
```

```
}
```

```
int fun2(int i)
```

```
{
```

```
 return fun1() * 5;
```

```
}
```

Run on IDE

Article compiled by **Venki**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

---

## Introduction to Reentrancy

### Introduction to Reentrancy

Virtually every embedded system uses interrupts; many support multitasking or multithreaded operations. These sorts of applications can expect the program's control flow to change contexts at just about any time. When that interrupt comes, the current operation gets put on hold and another function or task starts running. What happens if functions and tasks share variables? Disaster surely looms if one routine corrupts another's data.

By carefully controlling how data is shared, we create reentrant functions, those that allow multiple concurrent invocations that do not interfere with each other. The word pure is sometimes used interchangeably with reentrant.

Like so many embedded concepts, reentrancy came from the mainframe era, in the days when memory was a valuable commodity. In those days compilers and other programs were often written to be reentrant, so a single copy of the tool lived in memory, yet was shared by perhaps a hundred users. Each person had his or her own data area, yet everyone running the compiler quite literally executed the identical code. As the operating system changed contexts from user to user it swapped data areas so one person's work didn't effect any other. Share the code, but not the data.

In the embedded world a routine must satisfy the following conditions to be reentrant:

1. It uses all shared variables in an atomic way, unless each is allocated to a specific instance of the function.
2. It does not call non-reentrant functions.
3. It does not use the hardware in a non-atomic way.

Quite a mouthful! Let's look at each of these in more detail.

## Atomic variables

Both the first and last rules use the word atomic, which comes from the Greek word meaning indivisible. In the computer world, atomic means an operation that cannot be interrupted. Consider the assembly language instruction:

**mov ax,bx**

Since nothing short of a reset can stop or interrupt this instruction, it's atomic. It will start and complete without any interference from other tasks or interrupts

The first part of Rule 1 requires the atomic use of shared variables. Suppose two functions each share the global variable foobar. Function A contains:

**temp = foobar;**

**temp += 1;**

**foobar = temp;**

This code is not reentrant, because foobar is used non-atomically. That is, it takes three statements to change its value, not one. The foobar handling is not indivisible; an interrupt can come between these statements and switch context to the other function, which then may also try and change foobar. Clearly there's a conflict; foobar will wind up with an incorrect value, the autopilot will crash, and hundreds of screaming people will wonder "why didn't they teach those developers about reentrancy?"

Suppose, instead, Function A looks like:

```
foobar += 1;
```

Now the operation is atomic, right? An interrupt cannot suspend processing with foobar in a partially changed state, so the routine is reentrant.

Except... do you really know what your C compiler generates? On an x86 processor that statement might compile to:

```
mov ax,[foobar]
```

```
inc ax
```

```
mov [foobar],ax
```

which is clearly not atomic, and so not reentrant. The atomic version is:

```
inc [foobar]
```

The moral is to be wary of the compiler. Assume it generates atomic code and you may find "60 Minutes" knocking at your door.

The second part of the first reentrancy rule reads "...unless each is allocated to a specific instance of the function." This is an exception to the atomic rule that skirts the issue of shared variables.

An "instance" is a path through the code. There's no reason a single function can't be called from many other places. In a multitasking environment, it's quite possible that several copies of the function may indeed be executing concurrently. (Suppose the routine is a driver that retrieves data from a queue; many different parts of the code may want queued data more or less simultaneously). Each execution path is an "instance" of the code. Consider:

```
int foo;
void some_function(void) {
 foo++;
}
```

foo is a global variable whose scope exists outside that of the function. Even if no other routine uses foo, some\_function can trash the variable if more than one instance of it runs at any time.

C and C++ can save us from this peril. Use automatic variables. That is, declare foo inside of the function. Then, each instance of the routine will use a new version of foo created from the stack, as follows:

```
void some_function(void) {
 int foo;
 foo++;
}
```

Another option is to dynamically assign memory (using malloc), again so each incarnation uses a unique data area. The fundamental reentrancy problem is thus avoided, as it's impossible for multiple instances to modify a common version of the variable.

## Two more rules

The other rules are very simple. Rule 2 tells us a calling function inherits the reentrancy problems of the callee. That makes sense. If other code inside the function trashes shared variables, the system is going to crash. Using a compiled language, though, there's an insidious problem. Are you sure-really sure-that all of the runtime library functions are reentrant? Obviously, string operations and a lot of other complicated things make library calls to do the real work. An awful lot of compilers also generate runtime calls to do, for instance, long math, or even integer multiplications and divisions.

If a function must be reentrant, talk to the compiler vendor to ensure that the entire runtime package is pure. If you buy software packages (like a protocol stack) that may be called from several places, take similar precautions to ensure the purchased routines are also reentrant.

Rule 3 is a uniquely embedded caveat. Hardware looks a lot like a variable; if it takes more than a single I/O operation to handle a device, reentrancy problems can develop.

Consider Zilog's SCC serial controller. Accessing any of the device's internal registers requires two steps: first write the register's address to a port, then read or write the register from the same port, the same I/O address. If an interrupt fires between setting the port and accessing the register, another function might take over and access the device. When control returns to the first function, the register address you set will be incorrect.

## Keeping code reentrant

What are our best options for eliminating non-reentrant code? The first rule of thumb is to avoid shared variables. Globals are the source of endless debugging woes and failed code. Use automatic variables or dynamically allocated memory.

Yet globals are also the fastest way to pass data around. It's not always possible to entirely eliminate them from real time systems. So, when using a shared resource (variable or hardware) we must take a different sort of action.

The most common approach is to disable interrupts during non-reentrant code. With interrupts off, the system suddenly becomes a single-process environment. There will be no context switches. Disable interrupts, do the non-reentrant work, and then turn interrupts back on.

Shutting interrupts down does increase system latency, reducing its ability to respond to external events in a timely manner. A kinder, gentler approach is to use a mutex (also known as binary semaphore) to indicate when a resource is busy. Mutexes are simple on-off state indicators whose processing is inherently atomic. These are often used as "in-use" flags to have tasks idle when a shared resource is not available.

Nearly every commercial real-time operating system includes mutexes. If this is your way of achieving reentrant code, by all means use an RTOS.

## Recursion

No discussion of reentrancy is complete without mentioning recursion, if only because there's so much confusion between the two.

A function is recursive if it calls itself. That's a classic way to remove iteration from many sorts of algorithms. Given enough stack space, this is a perfectly valid--though tough to debug--way to write code. Since a recursive function calls itself, clearly it must be reentrant to avoid trashing its variables. So all recursive functions must be reentrant, but not all reentrant functions are recursive.

**Jack G. Ganssle** is a lecturer and consultant on embedded development issues, and a regular contributor to *Embedded Systems Programming*. Contact him at [jack@ganssle.com](mailto:jack@ganssle.com).

## Resources

Greenberg, Kenneth F. "Sharing Your Code," *Embedded Systems Programming*, August 1990, p 40.

Barr, Michael. "[Choosing a Compiler: The Little Things](#)", *Embedded Systems Programming*, May 1999, p. 71.

Simon, David. *An Embedded Software Primer*. Reading, MA: Addison-Wesley, 1999.

---

**When is a function reentrant? How does that relate to it being thread-safe?**

### When is a function reentrant?

A function is **reentrant** if it can be invoked while already in the process of executing. That is, a function is reentrant if it can be interrupted in the middle of execution (for example, by a signal or interrupt) and invoked again before the interrupted execution completes.

For example, the following function is *not* reentrant, because the observed value of the summation depends on when and where the function is interrupted (or, in the case of multithreading, how two or more threads race into the function):

```
1. static int sum = 0;
```

```
2.
3. int increment(int i) {
4. sum += i;
5. return sum;
6. }
```

We can make this function reentrant by making the sum not a global variable and instead requiring the caller to maintain it:

```
1. int increment(int sum, int i) {
2. return sum + i;
3. }
```

`ctime()`, `gmtime()`, and `strtok()` are examples of functions in Standard C that are (or at least often are) not reentrant. Later versions of the standard have created reentrant versions, for example `strtok_r()`, with the primary difference that the caller passes in storage instead of the function maintaining it globally, as above.

### How does that relate to it being thread-safe?

The term reentrancy predates threads and is often used only in a single threaded context, for example when discussing if a function is signal or interrupt-safe. *Reentrancy and thread-safety are not the same and should not be confused.*

Although a reentrant function is more likely to be thread-safe all else equal, a reentrant function **need not** be thread-safe, and a thread-safe function **need not** be reentrant. For example, reentrant functions that operate on the same data (same inputs) may not be thread-safe. Similarly, a thread-safe function that achieves safety through a mutex is potentially not reentrant, as the interruption could deadlock on the already-acquired mutex.

To summarize, we might define reentrancy and thread safety in similar terms, but with slightly different semantics:

A *reentrant function* can be invoked, interrupted, and re-invoked. Such a function can be invoked simultaneously by multiple threads if and only if each invocation references or provides unique data and inputs.

A *thread-safe function* can be invoked simultaneously by multiple threads, even if each invocation references or provides the same data or input, as all access is serialized.

### Writing reentrant code.

Writing code that is reentrant is not hard. The styles and patterns adapted by programmers since the mid-1990s are largely reentrant. Indeed, most programmers today balk at solutions to problems such as the approach adopted by `strtok()`, even if not concerned with reentrancy or thread safety.

Basic guidelines:

- Do not access mutable global or function-static variables.
- Do not self-modify code.
- Do not invoke another function that is itself non-reentrant.

88888888

ikipedia gives us good definitions:

A piece of code is **thread-safe** if it only manipulates shared data structures in a manner that guarantees safe execution by multiple threads at the same time.  
(see [Thread safety](#))

In [computing](#), a [computer program](#) or [subroutine](#) is called **reentrant** if it can be interrupted in the middle of its execution and then safely called again ("re-entered") before its previous invocations complete execution. (see [Reentrancy \(computing\)](#))

Details, including examples can be found on the Wikipedia pages.

---

## Is malloc reentrant?

No it is not. malloc doesn't let you reenter it. malloc uses locks(to make it thread safe).

Note: re-entrant routines cannot have locks.

Here is a snippet to help you understand

```
malloc(); //initial call
lock(memory_lock); //acquire lock inside malloc implementation
signal_handler(); //interrupt and process signal
malloc(); //call malloc() inside signal handler
lock(memory_lock); //try to acquire lock in malloc implementation
// DEADLOCK! We wait for release of memory_lock, but
// it won't be released because the original malloc call is interrupted
```

so to summarize ,  
malloc uses locks .re-entrant doesn't use locks. This proves malloc is not re-entrant

---

## What are thread safe functions?

rom wikipedia :

*Thread safety is a computer programming concept applicable in the context of multi-threaded programs. A piece of code is thread-safe if it functions correctly during simultaneous execution*

by multiple threads. In particular, it must satisfy the need for multiple threads to access the same shared data, and the need for a shared piece of data to be accessed by only one thread at any given time.

...

There are a few ways to achieve thread safety:

### Re-entrancy

Writing code in such a way that it can be partially executed by one task, reentered by another task, and then resumed from the original task. This requires the saving of state information in variables local to each task, usually on its stack, instead of in static or global variables.

### Mutual exclusion

Access to shared data is serialized using mechanisms that ensure only one thread reads or writes the shared data at any time. Great care is required if a piece of code accesses multiple shared pieces of data—problems include race conditions, deadlocks, livelocks, starvation, and various other ills enumerated in many operating systems textbooks.

### Thread-local storage

Variables are localized so that each thread has its own private copy. These variables retain their values across subroutine and other code boundaries, and are thread-safe since they are local to each thread, even though the code which accesses them might be reentrant.

### Atomic operations

Shared data are accessed by using atomic operations which cannot be interrupted by other threads. This usually requires using special machine language instructions, which might be available in a runtime library. Since the operations are atomic, the shared data are always kept in a valid state, no matter what other threads access it. Atomic operations form the basis of many thread locking mechanisms.

**read more :**

[Thread safety](#)

---

## Thread safety

From Wikipedia, the free encyclopedia



This article may require [cleanup](#) to meet Wikipedia's [quality standards](#). The specific problem is: **the opening definition is tautological/circular** Please help [improve this article](#) if you can. (January 2016) ([Learn how and when to remove this template message](#))

**Thread safety** is a [computer programming](#) concept applicable to [multi-threaded](#) code. Thread-safe code only manipulates shared data structures in a manner that ensures that all threads behave properly and fulfil their design specifications without unintended interaction. There are various strategies for making thread-safe data structures.<sup>[1][2]</sup>

A program may execute code in several threads simultaneously in a shared [address space](#) where each of those threads has access to virtually all of the [memory](#) of every other thread. Thread safety is a property that allows code to run in multithreaded environments by re-establishing some of the correspondences between the actual flow of control and the text of the program, by means of [synchronization](#).

### Contents

[\[hide\]](#)

- [1 Levels of thread safety](#)
- [2 Implementation approaches](#)
- [3 Examples](#)
- [4 See also](#)
- [5 References](#)
- [6 External links](#)

## Levels of thread safety[\[edit\]](#)

[Software libraries](#) can provide certain thread-safety guarantees. For example, concurrent reads might be guaranteed to be thread-safe, but concurrent writes might not be. Whether a program using such a library is thread-safe depends on whether it uses the library in a manner consistent with those guarantees.

Different vendors use slightly different terminology for thread-safety:[\[3\]](#)[\[4\]](#)[\[5\]](#)[\[6\]](#)

- **Thread safe:** Implementation is guaranteed to be free of [race conditions](#) when accessed by multiple threads simultaneously.
- **Conditionally safe:** Different threads can access different objects simultaneously, and access to shared data is protected from race conditions.
- **Not thread safe:** Code should not be accessed simultaneously by different threads.

Thread safety guarantees usually also include design steps to prevent or limit the risk of different forms of [deadlocks](#), as well as optimizations to maximize concurrent performance. However, deadlock-free guarantees can not always be given, since deadlocks can be caused by [callbacks](#) and violation of [architectural layering](#) independent of the library itself.

## Implementation approaches[\[edit\]](#)

Below we discuss two approaches for avoiding [race conditions](#) to achieve thread safety.

**The first class of approaches focuses on avoiding shared state, and includes:**

### Re-entrancy

Writing code in such a way that it can be partially executed by a thread, reexecuted by the same thread or simultaneously executed by another thread and still correctly complete the original execution. This requires the saving of [state](#) information in variables local to each execution, usually on a stack, instead of in [static](#) or [global](#) variables or other non-local state. All non-local state must be accessed through atomic operations and the data-structures must also be reentrant.

### Thread-local storage

Variables are localized so that each thread has its own private copy. These variables retain their values across [subroutine](#) and other code boundaries, and are thread-safe since they are local to each thread, even though the code which accesses them might be executed simultaneously by another thread.

### Immutable objects

The state of an object cannot be changed after construction. This implies both that only read-only data is shared and that inherent thread safety is attained. Mutable (non-const) operations can

then be implemented in such a way that they create new objects instead of modifying existing ones. This approach is used by the *string* implementations in Java, C# and Python.<sup>[7]</sup>

**The second class of approaches are synchronization-related, and are used in situations where shared state cannot be avoided:**

### Mutual exclusion

Access to shared data is *serialized* using mechanisms that ensure only one thread reads or writes to the shared data at any time. Incorporation of mutual exclusion needs to be well thought out, since improper usage can lead to side-effects like [deadlocks](#), [livelocks](#) and [resource starvation](#).

### Atomic operations

Shared data are accessed by using atomic operations which cannot be interrupted by other threads. This usually requires using special [machine language](#) instructions, which might be available in a [runtime library](#). Since the operations are atomic, the shared data are always kept in a valid state, no matter how other threads access it. Atomic operations form the basis of many thread locking mechanisms, and are used to implement mutual exclusion primitives.

## Examples[edit]

In the following piece of [Java](#) code, the method is thread-safe:

```
class Counter {
 private int i = 0;

 public synchronized void inc() {
 i++;
 }
}
```

In the [C programming language](#), each thread has its own stack. However, a [static variable](#) is not kept on the stack; all threads share simultaneous access to it. If multiple threads overlap while running the same function, it is possible that a static variable might be changed by one thread while another is midway through checking it. This difficult-to-diagnose [logic error](#), which may compile and run properly most of the time, is called a [race condition](#). One common way to avoid this is to use another shared variable as a "[lock](#)" or "[mutex](#)" (from [mutual exclusion](#)).

In the following piece of C code, the function is thread-safe, but not reentrant:

```
include <pthread.h>

int increment_counter ()
{
 static int counter = 0;
 static pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

```
// only allow one thread to increment at a time
pthread_mutex_lock(&mutex);

++counter;

// store value before any other threads increment it further
int result = counter;

pthread_mutex_unlock(&mutex);

return result;
}
```

In the above, `increment_counter` can be called by different threads without any problem since a mutex is used to synchronize all access to the shared `counter` variable. But if the function is used in a reentrant interrupt handler and a second interrupt arises inside the function, the second routine will hang forever. As interrupt servicing can disable other interrupts, the whole system could suffer.

The same function can be implemented to be both thread-safe and reentrant using the lock-free [atomics](#) in C++11:

```
include <atomic>

int increment_counter ()
{
 static std::atomic<int> counter(0);

 // increment is guaranteed to be done atomically
 int result = ++counter;

 return result;
}
```