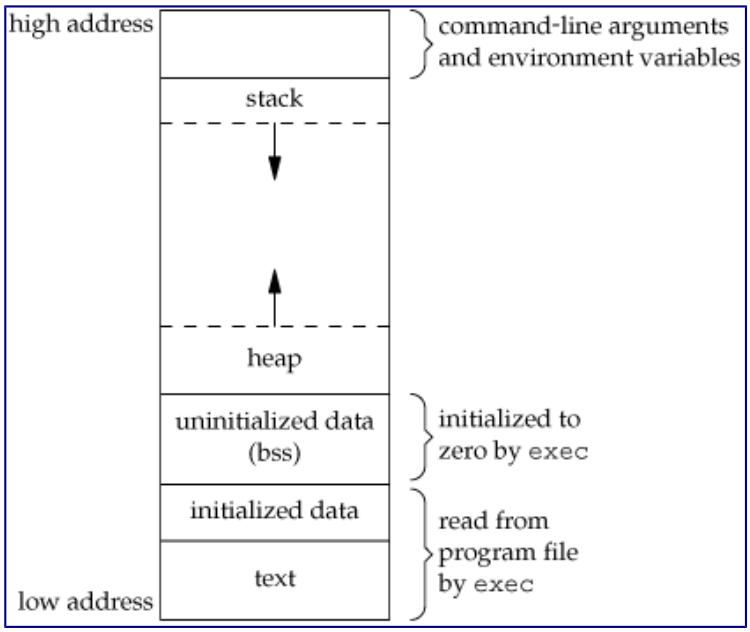
Memory Layout of C Programs

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

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



A typical memory layout of a running process

1. Text Segment:

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

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

the text segment is often ***read-only.***

***2.*Initialized Data Segment:**

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

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

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

For instance the global string defined by char s[] = “hello world” in C and a C statement like int debug=1 outside the main (i.e. global) would be stored ***in initialized read-write area***.

And a global C statement like const char* string = “hello world” makes the string literal “hello world” to be stored in ***initialized read-only area*** and the character pointer variable string in initialized **read-write** area.

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

3. Uninitialized Data Segment:

Uninitialized data segment, often called the “bss” segment, named after an ancient assembler operator that stood for “block started by symbol.” Data in this segment is initialized by the kernel to arithmetic 0 before the program starts executing

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

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

4. Stack:

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

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

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.

Storage Classes in C

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

C language uses 4 storage classes, namely:

auto: This is the default storage class for all the variables declared inside a function or a block. Hence, the keyword auto is rarely used while writing programs in C language.

Auto variables can be only accessed within the block/function they have been declared and not outside them (which defines their scope). Of course, these can be accessed within nested blocks within the parent block/function in which the auto variable was declared.

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

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

This basically signifies that we are not initializing a new variable but instead we are using/accessing the global variable only. The main purpose of using extern variables is that they can be accessed between two different files which are part of a large program. For more information on how extern variables work, have a look at this link.

static: This storage class is used to declare static variables which are popularly used while writing programs in C language. Static variables have a property of preserving their value even after they are out of their scope! Hence, static variables preserve the value of their last use in their scope. So we can say that they are initialized only once and exist till the termination of the program.

Thus, no new memory is allocated because they are not re-declared. Their scope is local to the function to which they were defined. Global static variables can be accessed anywhere in the program. By default, they are assigned the value 0 by the compiler.

register: This storage class declares register variables which have the same functionality as that of the auto variables. The only difference is that the compiler tries to store these variables in the register of the microprocessor if a free register is available. This makes the use of register variables to be much faster than that of the variables stored in the memory during the runtime of the program. If a free register is not available, these are then stored in the memory only.

Usually few variables which are to be accessed very frequently in a program are declared with the register keyword which improves the running time of the program. An important and interesting point to be noted here is that we cannot obtain the address of a register variable using pointers.

```c
	// A C program to demonstrate different storage
	// classes
#include <stdio.h>

	// declaring and initializing an extern variable
extern int x = 9;

	// declaring and initialing a global variable z
	// simply int z; would have initialized z with
	// the default value of a global variable which is 0
int z = 10;

int main()
{
		// declaring an auto variable (simply
		// writing "int a=32;" works as well)
	auto int a = 32;

		// declaring a register variable
	register char b = 'G';

		// telling the compiler that the variable
		// x is an extern variable and has been
		// defined elsewhere (above the main
		// function)
	extern int z;

	printf("Hello World!\n");

		// printing the auto variable 'a'
	printf("\nThis is the value of the auto integer 'a': %d\n",a);

		// printing the extern variables 'x'
		// and 'z'
	printf("\nThese are the values of the extern integers 'x' and
'z'respectively: %d and %d\n", x, z);

		// printing the register variable 'b'
	printf("\nThis is the value of the register character 'b':
%c\n",b);

		// value of extern variable x modified
	x = 2;

		// value of extern variable z modified
	z = 5;

		// printing the modified values of
		// extern variables 'x' and 'z'
	printf("\nThese are the modified values of the extern integers
'x' and'z' respectively: %d and %d\n",x,z);
```

```
          // using a static variable 'y'
     printf("\n'y' is a static variable and its value is NOT
initialized to 5 after the first iteration! See for yourself
:\n");

     while (x > 0)
     {
     static int y = 5;
     y++;

          // printing value of y at each iteration
     printf("The value of y is %d\n",y);
     x--;
}
          // exiting
     printf("\nBye! See you soon. :)\n");
     return 0;
}
```

Q. What are the uses of the keyword static?

Answer :

This simple question is rarely answered completely. Static has two distinct uses in C:

(a) A variable declared static within the body of a function maintains its value between function invocations.

(b) A variable declared static within a file, (but outside the body of a function) is accessible by all functions within that file. It is not accessible by functions within any other module. That is, it is a localized global.

Q. What does the keyword volatile mean? Give three different examples of its use.

A volatile variable is one that can change unexpectedly. Consequently, the compiler can make no assumptions about the value of the variable. In particular, the optimizer must be careful to reload the variable every time it is used instead of holding a copy in a register. Examples of volatile variables are:

(a) Hardware registers in peripherals (e.g., status registers)

(b) Non-stack variables referenced within an interrupt service routine.

(c) Variables shared by multiple tasks in a multi-threaded application.

Q. Can a parameter be both const and volatile? Explain your answer.

Answer :

Yes. An example is a read only status register. It is volatile because it can change unexpectedly. It is const because the program should not attempt to modify it.

Q. What is void pointer explain with example and why we use it ?

Answer :

Its is also called generic pointer used to point an unspecified object.

```
int main()
{
    int i = 6;
    char c = 'a';
    void *generic;

    generic = &i;
    printf("generic points to the integer value %d\n", *(int*) generic); // need to do typecast

    generic = &c;
    printf("generic now points to the character %c\n", *(char*) generic); // need to do typecast

    return 0;
}
```

One common application of void pointers is creating functions which take any kind of pointer as an argument and perform some operation on the data which doesn't depend on the data contained. The perfect example of void pointer is

```
void * memset ( void * ptr, int value, size_t num );
```

Q. What is function pointer ? explain it with practical example !

A function pointer is a variable which is used to hold the starting address of a functions and the same can be used to invoke a function. Function pointers do always point to a function having a specific type so, all functions used with the same function pointer must have the same parameters and return type.

```
#include <stdio.h>

// function prototypes

int add(int x, int y);

int subtract(int x, int y);

int domath(int (*mathop)(int, int), int x, int y);

// add x + y

int add(int x, int y)

{

        return x + y;

}

// subtract x - y

int subtract(int x, int y)

 {
```

```
        return x - y;
}
// run the function pointer with inputs
int domath(int (*mathop)(int, int), int x, int y) {
return (*mathop)(x, y);
}
// calling from main
int main()
{
        // call math function with add
        int a = domath(add, 10, 2);
        printf("Add gives: %d\n", a);
        // call math function with subtract
        int b = domath(subtract, 10, 2);
        printf("Subtract gives: %d\n", b);
}
```

Q. Swap the nibble of this single byte 0xAB ?

Answer :

```
/* Less Complex way */
x = ( (byte & 0x0F) << 4 ); // Lower nibble comes out & shifted left by nibble
y = ( (byte & 0xF0) >> 4 ); // Higher nibble comes out & shifted right by nibble
byte = x | y;
```

or

```
/* Efficient way */
byte = ( (byte & 0x0F) << 4) | ( (byte & 0xF0) >> 4 )
```

Q. Given an integer variable a, write two code fragments. The first should set bit 3 of a. The second should clear bit 3 of a. In both cases, the remaining bits should be unmodified.

Answer :

Use #defines and bit masks. This is a highly portable method, and is the one that should be used. My optimal solution to this problem would be:

```
/* Bit stuff */
```

```
#define BIT3 (1 << 3)

static int a;

void set_bit3(void)

{

    a |= BIT3;

}

void clear_bit3(void)

{

    a &= ~BIT3;

}
```

1.Write a Macro’s Set,clear and toggle n’th bit using bit wise operators?

```
#include<stdio.h>
void bin(unsigned n);
#define SetBit(number,bit) (number|=(1<<bit-1))
#define ClearBit(number,bit) (number&=(~(1<<bit-1)))
#define Toggle(number,bit) (number ^ (1<<bit-1))
int main()
{
        int i=7;
        printf("\n Binary of 7 is      : ");
        bin(i);
        i=SetBit(i,4);
        printf("\n\n Set 4th bit of 7    : ");
        bin(i);
        i=ClearBit(i,4);
        printf("\n\n Clear 4th bit of 7  : ");
        bin(i);
        i=Toggle(i,1);
        printf("\n\n Toggle 1st bit of 7 : ");
        bin(i);
        return 0;
}
void bin(unsigned n)
{
   unsigned i;
   for (i = (1 << 31) ; i > 0; i = (i>>1))
```

```
        if(n & i)
                        printf("1");
                else
                        printf("0");
}
```

2.Write a c program to implement XOR functionality with out using XOR(^) operator?

```
#include <stdio.h>
inline int Xor(int x, int y)
{
        return ( (~x) & y ) | ( x & (~y) );
}

int main( )
{
        printf("%d",Xor(1,3));              // Answer would be 2 and it run successful
        return 0;
}
```

3.Count number of 1s in given binary number

```
#include<stdio.h>
/* 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;
}

/* Program to test function countSetBits */
int main()
{
   int i = 9;
```

```
    printf("%d", countSetBits(i));
    getchar();
    return 0;
}
```

4.Swap two nibbles in a byte

```
#include <stdio.h>
unsigned char swapNibbles(unsigned char x)
{
    return ( (x & 0x0F)<<4 | (x & 0xF0)>>4 );
}

int main()
{
    unsigned char x = 0xAB;             //Assigned in single byte hexadecimal value
    printf("%x", swapNibbles(x));
    return 0;
}
```

5.How to turn off a particular bit in a number?

To turn off the particular bit in given number we use bitwise ‘<<‘, ‘&’ and ‘~’ operators. Using expression “~(1 << (k – 1))“, we get a number which has all bits set, except the k’th bit. If we do bitwise ‘&’ of this expression with n, we get a number which has all bits same as n except the k’th bit which is 0.

```
#include<stdio.h>
 // Returns a number that has all bits same
// except the bit which is made 0
int TurnOffParticularBit(int number, int bit)
{
    // bit must be greater than 0
    if (bit <= 0) return number;
   // Do & of number with a number with all set bits except
    // the bit
    return (number & ~(1 << (bit - 1)));
}

int main()
{
     printf("%d",TurnOffParticularBit(15, 4)  );
```

```
    return 0;
}
```

6.Check if a number is multiple of 9 using bitwise operators

Check if a number is multiple of 9 using bitwise operators

- The most simple way to check for n's divisibility by 9 is to do n%9.
- Another method is to sum the digits of n. If sum of digits is multiple of 9, then n is multiple of 9.
- The above methods are not bitwise operators based methods and require use of '%' and '/'. The **bitwise operators are generally faster than modulo and division** operators. Following is a bitwise operator based method to check divisibility by 9.

```
#include<iostream>
using namespace std;

// Bitwise operator based function to check multiple of 9
bool isMulOf9(int n)
{
   // Base cases
   if (n == 0 || n == 9)
      return true;
   if (n < 9)
      return false;
   // if the number is greater than 9, then do this
   return isMulOf9((int)(n>>3) - (int)(n&7));
}
int main()
{
   if( isMulOf9(18) )
        cout << "Given Number is multiple of 9 \n" << endl;
   else
        cout << "Given Number is not multiple of 9 \n" << endl;
  return 0;
}
```

7.How to write a running C code without main()?

```
/* Code without main( ) */
#include<stdio.h>
#define fun main
int fun(void)
```

```
{
   printf("www.firmcodes.com");
   return 0;
}
```

8.How to pass a 2D array as a parameter in C?

Method 1 –When second dimension is known

```
/* When second argument is known */
#include <stdio.h>
const int n = 3;
void print(int arr[][n], int m)
{
   int i, j;
   for (i = 0; i < m; i++)
     for (j = 0; j < n; j++)
       printf("%d ", arr[i][j]);
}

int main()
{
   int arr[][3] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
   print(arr, 3);
   return 0;
}
```

Method 2 – Using Single Pointer

```
/* Using Single Pointer */
#include <stdio.h>
void print(int *arr, int m, int n)
{
   int i, j;
   for (i = 0; i < m; i++)
     for (j = 0; j < n; j++)
       printf("%d ", *((arr+i*n) + j));
}
int main()
{
   int arr[][3] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
   int m = 3, n = 3;
   print((int *)arr, m, n);
   return 0;
```

```
}
```

Method 3 – Using an array of pointers or double pointer

```
/* Using an array of pointers or double pointer */
#include <stdio.h>
// Same as "void print(int **arr, int m, int n)"
void print(int *arr[], int m, int n)
{
    int i, j;
    for (i = 0; i < m; i++)
      for (j = 0; j < n; j++)
        printf("%d ", *((arr+i*n) + j));
}

int main()
{
    int arr[][3] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
    int m = 3;
    int n = 3;
    print((int **)arr, m, n);
    return 0;
}
```

9.Print a long int in C using putchar() only

Print a long int in C using putchar() only

Write a C function print(n) that takes a long int number n as argument, and prints it on console. The only allowed library function is putchar( ), no other function like itoa( ) or printf( ) is allowed. Use of loops is also not allowed.

Since putchar( ) prints a character, we need to call putchar( ) for all digits. Recursion can always be used to replace iteration, so using recursion we can print all digits one by one. Now the question is putchar( ) prints a character, how to print digits using putchar( ). We need to convert every digit to its corresponding ASCII value, this can be done by using ASCII value of '0'. Following is complete C program.

```
/* C program to print a long int number using putchar() only*/
#include <stdio.h>
void print(long n)
{
    // If number is smaller than 0, put a - sign and
```

```
    // change number to positive
    if (n < 0) {
        putchar('-');
        n = -n;
    }

    // If number is 0
    if (n == 0)
        putchar('0');
    // Remove the last digit and recur
    if (n/10)
        print(n/10);
    // Print the last digit
    putchar(n%10 + '0');
}

int main()
{
    long int n = 12045;
    print(n);
    return 0;
}
```

10.How to dynamically allocate a 2D array in C ?

Method 1 :-Using a single pointer

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int r = 3, c = 4;
    int *arr = (int *)malloc(r * c * sizeof(int));

    int i, j, count = 0;
    for (i = 0; i <  r; i++)
      for (j = 0; j < c; j++)
         *(arr + i*c + j) = ++count;

    for (i = 0; i <  r; i++)
      for (j = 0; j < c; j++)
         printf("%d ", *(arr + i*c + j));
```

```
   /* Code for further processing and free the
      dynamically allocated memory */
  return 0;
}
```

Method 2 :-Using an array of pointers

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int r = 3, c = 4, i, j, count;
    int *arr[r];
    for (i=0; i<r; i++)
        arr[i] = (int *)malloc(c * sizeof(int));

    // Note that arr[i][j] is same as *(*(arr+i)+j)
    count = 0;
    for (i = 0; i <  r; i++)
      for (j = 0; j < c; j++)
        arr[i][j] = ++count; // Or *(*(arr+i)+j) = ++count

    for (i = 0; i <  r; i++)
      for (j = 0; j < c; j++)
        printf("%d ", arr[i][j]);

    /* Code for further processing and free the
      dynamically allocated memory */
   return 0;
}
```

Method 3 :-Using pointer to a pointer

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int r = 3, c = 4, i, j, count;
    int **arr = (int **)malloc(r * sizeof(int *));
    for (i=0; i<r; i++)
```

```
        arr[i] = (int *)malloc(c * sizeof(int));
    // Note that arr[i][j] is same as *(*(arr+i)+j)
    count = 0;
    for (i = 0; i <  r; i++)
      for (j = 0; j < c; j++)
        arr[i][j] = ++count;  // OR *(*(arr+i)+j) = ++count
    for (i = 0; i <  r; i++)
      for (j = 0; j < c; j++)
        printf("%d ", arr[i][j]);

   /* Code for further processing and free the
      dynamically allocated memory */
   return 0;
}
```

1. Find position of the only set bit

```
/* C program to find position of only set bit in a given number */
#include <stdio.h>
// A utility function to check whether n is power of 2 or not
int isPowerOfTwo(unsigned n)
{  return n && (! (n & (n-1)) ); }
// Returns position of the only set bit in 'n'
int findPosition(unsigned n)
{
    if (!isPowerOfTwo(n))
        return -1;
   unsigned count = 0;
    // One by one move the only set bit to right till it reaches end
    while (n)
    {
        n = n >> 1;
         // increment count of shifts
        ++count;
    }
     return count;
}

int main(void)
{
```

```
    int n = 0;
    int pos = findPosition(n);
    (pos == -1)? printf("n = %d, Invalid number\n", n):
                 printf("n = %d, Position %d \n", n, pos);

    n = 12;
    pos = findPosition(n);
    (pos == -1)? printf("n = %d, Invalid number\n", n): printf("n = %d, Position %d
\n", n, pos);
     n = 128;
    pos = findPosition(n);
    (pos == -1)? printf("n = %d, Invalid number\n", n):printf("n = %d, Position %d \n",
n, pos);
     return 0;
}
```

2. Write your own strcmp

Following are the detailed steps.

1) Iterate through every character of both strings and do following for each character.

a)If str1[i] is same as str2[i], then continue.

b) If inverting the 6th least significant bit of str1[i] makes it same as str2[i], then continue. For example, if str1[i] is 65, then inverting the 6th bit will make it 97. And if str1[i] is 97, then inverting the 6th bit will make it 65.

c) If any of the above two conditions is not true, then break.

2) Compare the last (or first mismatching in case of not same) characters.

```
/* Implement the strcmp function  */
#include <stdio.h>
/* implementation of strcmp that ingnores cases */
int ic_strcmp(char *s1, char *s2)
{
    int i;
    for (i = 0; s1[i] && s2[i]; ++i)
    {
        /* If characters are same or inverting the 6th bit makes them same */
        if (s1[i] == s2[i] || (s1[i] ^ 32) == s2[i])
            continue;
        else
```

```
            break;
    }

    /* Compare the last (or first mismatching in case of not same) characters */
    if (s1[i] == s2[i])
        return 0;
    if ((s1[i]|32) < (s2[i]|32)) //Set the 6th bit in both, then compare
        return -1;
    return 1;
}

int main(void)
{
    printf("ret: %d\n", ic_strcmp("", "c/c++"));
    printf("ret: %d\n", ic_strcmp("firmcodes", "firmware"));
    printf("ret: %d\n", ic_strcmp("microcontrollers", "linux"));

    return 0;
}
```

3. Binary representation of a given number

```
/* Binary representation of number */
#include<stdio.h>
void bin(unsigned n)
{
    unsigned i;
    for (i = (1 << 31) ; i > 0; i = (i>>1))
        if(n & i)
                        printf("1");
                else
                        printf("0");
}

int main(void)
{
    bin(7);
}
```

4. Add two numbers without using arithmetic operators

Add two numbers without using arithmetic operators

Write a function Add( ) that returns sum of two integers. The function should not use any of the arithmetic operators (+, ++, –, -, .. etc).

Sum of two bits can be obtained by performing XOR (^) of the two bits. Carry bit can be obtained by performing AND (&) of two bits.
Above is simple Half Adder logic that can be used to add 2 single bits. We can extend this logic for integers. If x and y don't have set bits at same position(s), then bitwise XOR (^) of x and y gives the sum of x and y. To incorporate common set bits also, bitwise AND (&) is used. Bitwise AND of x and y gives all carry bits. We calculate (x & y) << 1 and add it to x ^ y to get the required result.

```
/* Add two numbers without using arithmetic operators */
#include<stdio.h>
int Add(int x, int y)
{
   // Iterate till there is no carry
   while (y != 0)
   {
      // carry now contains common set bits of x and y
      int carry = x & y;

      // Sum of bits of x and y where at least one of the bits is not set
      x = x ^ y;

      // Carry is shifted by one so that adding it to x gives the required sum
      y = carry << 1;
   }
   return x;
}

int main()
{
   printf("%d", Add(15, 32));
   return 0;
}
```

5. Smallest of three integers without comparison operators

Method 1: –Repeated Subtraction

Take a counter variable c and initialize it with 0. In a loop, repeatedly subtract x, y and z by 1 and increment c. The number which becomes 0 first is the smallest. After the loop terminates, c will hold the minimum of 3.

```
#include<stdio.h>
int smallest(int x, int y, int z)
{
  int c = 0;
  while ( x && y && z )
  {
     x--;  y--; z--; c++;
  }
  return c;
}

int main()
{
   int x = 12, y = 15, z = 5;
   printf("Minimum of 3 numbers is %d", smallest(x, y, z));
   return 0;
}
```

Drawback:-This methid doesn't work for negative numbers

Method 2: –Use Bit Operations

```
#include<stdio.h>
#define CHAR_BIT 8
/*Function to find minimum of x and y*/
int min(int x, int y)
{
  return  y + ((x - y) & ((x - y) >>
           (sizeof(int) * CHAR_BIT - 1)));
}

/* Function to find minimum of 3 numbers x, y and z*/
```

```
int smallest(int x, int y, int z)
{
    return min(x, min(y, z));
}

int main()
{
   int x = 12, y = 15, z = 5;
   printf("Minimum of 3 numbers is %d", smallest(x, y, z));
   return 0;
}
```

Method 3: –Use Division operator

We can also use division operator to find minimum of two numbers. If value of (a/b) is zero, then b is greater than a, else a is greater

```
#include <stdio.h>

// Using division operator to find minimum of three numbers
int smallest(int x, int y, int z)
{
    if (!(y/x))  // Same as "if (y < x)"
        return (!(y/z))? y : z;
    return (!(x/z))? x : z;
}

int main()
{
    int x = 78, y = 88, z = 68;
    printf("Minimum of 3 numbers is %d", smallest(x, y, z));
    return 0;
}
```

6. What is the best way in C to convert a number to a string?

```
/* Convert number into string */
#include<stdio.h>
int main()
{
    char result[50];
    float num = 23.34;
    sprintf(result, "%f", num);
```

```
    printf("\n The string for the num is %s", result);
    getchar();
}
```

7. Print “Even” or “Odd” without using conditional statement

```
#include<stdio.h>

int main()
{
    int no;
    printf("Enter a no: ");
    scanf("%d", &no);
    (no & 1 && printf("odd"))|| printf("even");
    return 0;
}

/* Main Logic */
(no & 1 && printf("odd"))|| printf("even");
//(no & 1)     <----------First expression
```

Let us understand First expression “no && 1”, if you enter 5 in place of no, your computer will store 5 as in binary like 0000 0000 0000 0101, with this binary you will perform & operation with 0000 0000 0000 0001 (which is 1 in decimal), in result, you get 0000 0000 0000 0001 (which is 1 in decimal), So output of first expression would be 1.

- Output of First expression only become 1 or 0 in every case.
- If output of first expression is 1, then it will perform && operation with printf(“odd”) and it will print ‘odd’.
- If output of first expression is 0, then it will perform && operation with printf(“odd”) which become false because of zero and than || operation is performed with printf(“even”) which will print ‘even’.

8. Interesting Facts about Macros and Preprocessors in C/C++

Interesting Facts about Macros and Preprocessors in C/C++

In a C/C++ program, all lines that start with ‘#’ are processed by preporcessor which is a special program invoked by the compiler. In a very basic term, preprocessor takes a C/C++ program and produces another C program without any ‘#’.

#include

When we use #include directive, the contents of included header file (after preprocessing) are copied to the current file.

Angular brackets < and > instruct the preprcessor to look in the standard folder where all header files are held. Double quotes “ and “ instruct the preprcessor to look into the current folder and if the file is not present in current folder, then in standard folder of all header files.

#define

When we use #define for a constant, the preprocessor produces a C program where the defined constant is searched and matching tokens are replaced with the given expression. For example in the following program max is defined as 100.

```
#include<stdio.h>

#define max 100

int main()

{

   printf("max is %d", max);

   return 0;

}

// Output: max is 100

// Note that the max inside "" is not replaced
```

Macro functions

The macros can take function like arguments, the arguments are not checked for data type. For example, the following macro INCREMENT(x) can be used for x of any data type.

```
#include <stdio.h>


#define INCREMENT(x) ++x
```

```
int main()
{
    char *ptr = "firmcodes";
    int x = 10;
    printf("%s  ", INCREMENT(ptr));
    printf("%d", INCREMENT(x));
    return 0;
}
// Output: firmcodes 11
```

Macro expansion

The macro arguments are not evaluated before macro expansion. For example consider the following program.

```
#include <stdio.h>

#define MULTIPLY(a, b) a*b

int main()
{
    // The macro is expended as 2 + 3 * 3 + 5, not as 5*8
    printf("%d", MULTIPLY(2+3, 3+5));
    return 0;
}
// Output: 16
```

Macro concatenation

The tokens passed to macros can be concatenated using operator ## called Token-Pasting operator.

```
#include <stdio.h>

#define merge(a, b) a##b

int main()
```

```
{
   printf("%d ", merge(12, 34));
}
// Output: 1234
```

Macro strings

A token passed to macro can be converted to a sting literal by using # before it.

```
#include <stdio.h>

#define get(a) #a

int main()
{
   printf("%s", get(firmcodes));
}
// Output: firmcodes
```

Multiline Macros

The macros can be written in multiple lines using ‘\’. The last line doesn’t need to have ‘\’.

```
#include <stdio.h>

#define PRINT(i, limit) while (i < limit) \
                        { \
                           printf("firmcodes"); \
                           i++; \
                        }

int main()
{
   int i = 0;
   PRINT(i, 3);
   return 0;
}
// Output: firmcodes  firmcodes firmcodes
```

Macro Argument Problems

The macros with arguments should be avoided as they cause problems sometimes. And Inline functions should be preferred as there is type checking parameter evaluation in inline functions. From C99 onward, inline functions are supported by C

language also. For example consider the following program. From first look the output seems to be 1, but it produces 36 as output.

```
#define square(x) x*x

int main()
{
  int x = 36/square(6); // Expended as 36/6*6
  printf("%d", x);
  return 0;
}
// Output: 36
```

If we use inline functions, we get the expected output. Also the program given above can be corrected using inline functions

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

int main()
{
  int x = 36/square(6);
  printf("%d", x);
  return 0;
}
// Output: 1
```

#if-else

Preprocessors also support if-else directives which are typically used for conditional compilation.

```
int main()
{
#if VERBOSE >= 2

    printf("Trace Message");

#endif
}
```

#ifndef

A header file may be included more than one time directly or indirectly, this leads to problems of redeclaration of same variables/functions. To avoid this problem, directives like defined, ifdef and ifndef are used.

```
#ifndef __HEADERFILE_H
```

```
#define __HEADERFILE_H
//Do some coding stuff here
#endif
//This will check for if HEADERFILE.h not defined, then define HEADERFILE, and then do some coding stuff
```

Some standard MACROS

There are some standard macros which can be used to print program file (__FILE__), Date of compilation (__DATE__), Time of compilation (__TIME__) and Line Number in C code (__LINE__).

```
#include <stdio.h>

int main()
{
  printf("Current File :%s\n", __FILE__ );
  printf("Current Date :%s\n", __DATE__ );
  printf("Current Time :%s\n", __TIME__ );
  printf("Line Number :%d\n", __LINE__ );
  return 0;
}

/* Output:
Current File :C:\Users\GfG\Downloads\deleteBST.c
Current Date :Feb 15 2014
Current Time :07:04:25
Line Number :8 */
```

9. C/C++ program to shutdown or turn off computer

C/C++ program to shutdown or turn off computer

This program turn off i.e shutdown your computer system. Firstly it will asks you to shutdown your computer if you press ‘y’ then your computer will shutdown in 30 seconds, system function of “stdlib.h” is used to run an executable file shutdown.exe which is present in C:\WINDOWS\system32 in Windows XP. You can use various options while executing shutdown.exe, you can use -t option to specify number of seconds after which shutdown occurs.

Syntax: shutdown -s -t x; here x is the number of seconds after which shutdown will occur.

By default shutdown occur after 30 seconds. To shutdown immediately you can write “shutdown -s -t 0”. If you wish to restart your computer then you can write “shutdown -r”.

Note:- If you are using Turbo C Compiler then execute your program from command prompt or by opening the executable file from the folder. Press F9 to build your executable file from source program. When you run program from within the compiler by pressing Ctrl+F9 it may not work.

Code for Windows XP

```
#include <stdio.h>
#include <stdlib.h>
 int main()
{
   char ch;
    printf("Do you want to shutdown your computer now (y/n)\n");
   scanf("%c", &ch);
    if (ch == 'y' || ch == 'Y')
      system("C:\\WINDOWS\\System32\\shutdown -s");
    return 0;
}
```

Code for Windows 7 & Above (Not Tested in Windows 10)

```
#include <stdio.h>
#include <stdlib.h>
 int main()
{
   system("C:\\WINDOWS\\System32\\shutdown /s");
    return 0;
}
```

To shutdown immediately use “C:\\WINDOWS\\System32\\ shutdown /s /t 0”. To restart use /r instead of /s. In above both program (XP & Win7), you need to be administrator otherwise this error message will appear “Permission denied” .

Code for Ubuntu Linux

```
#include <stdio.h>

 int main()

{

    system("shutdown -P now");

    return 0;

}
```

You need to be logged in as root user for above program to execute otherwise you will get the message shutdown: Need to be root, now specifies that you want to shutdown immediately. ‘-P’ option specifies you want to power off your machine. You can specify minutes as:

shutdown -P “number of minutes”

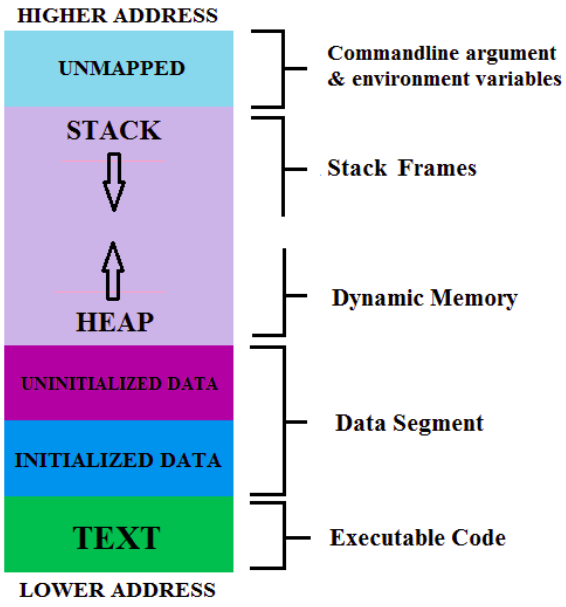
For more help or options type at terminal: man shutdown.

10. Memory Layout of C Program

What is memory layout of C-program ?

In practical words, when you run any C-program, its executable image is loaded into RAM of computer in an organized manner which is called process address space or Memory layout of C program.
This memeory layout is organized in following fashion:

- Text segment
- Data segment
- Heap segment
- Stack segment
- Unmapped or reserved



Text segment

Text segment contain executable instructions of your C program, its also called code segment. This is the machine language representation of the program steps to be carried out, including all functions making up the program, both user defined and system. The text segment is sharable so that only a single copy needs to be in memory for different executing programs, such as text editors, shells, and so on. Usually, text segment is read-only, to prevent a program from accidentally modifying its instructions.

Data segment :-There are two sub section of this segment called initialized & uninitialized data segment

Initialized data:- It contains both static and global data that are initialized with non-zero values.This segment can be further classified into *read-only* area and *read-write* area.

For example : The global string defined by char string[ ] = “hello world” and a statement like int count=1 outside the main (i.e. global) would be stored in initialized read-write area. And a global statement like const int A=3 makes the variable ‘A’ read-only and to be stored in initialized read-only area.

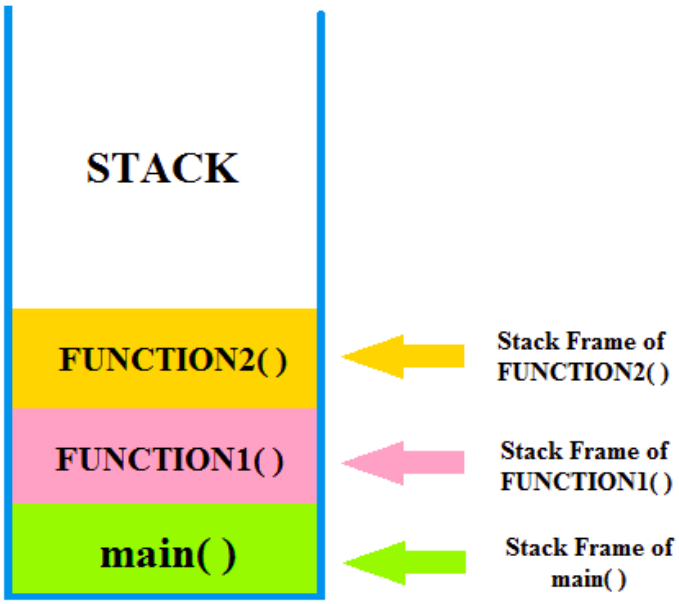
Uninitialized data (bss segment):- Uninitialized data segment is also called BSS segment. BSS stands for ‘Block Started by Symbol’ named after an ancient assembler operator. Uninitialized data segment contains all global and static variables that are initialized to zero or do not have explicit initialization in source code.

For example : The global variable declared as int A would be stored in uninitialized data segment. A statement like static int X=0 will also stored in this segment cause it initialized with zero.

Heap segment:- The heap segment is area where dynamically allocated memory (allocated by malloc(), calloc(), realloc() and new for C++) resides.

- When we allocate memory through dynamic allocation techniques(in simple word, run time memory allocation), program acquire space from system and process address space grows that's why we saw upward arrow indication in figure for Heap.
- We can free dynamically allocated memory space (by using Free() or delete). Freed memory goes back to the heap but doesn't have to be returned to system (it doesn't have to be returned at all), so unordered malloc/free's eventually cause *heap fragmentation.*
- When we use dynamic allocation for acquire memory space we must keep track of allocated memory by using its address.
- Memory leak error is cause by excess use of dynamic allocation or Heap fragmentation.

Stack segment



The stack segment is area where local variables are stored. By saying local variable means that all those variables which are declared in every function including main( ) in your C program.

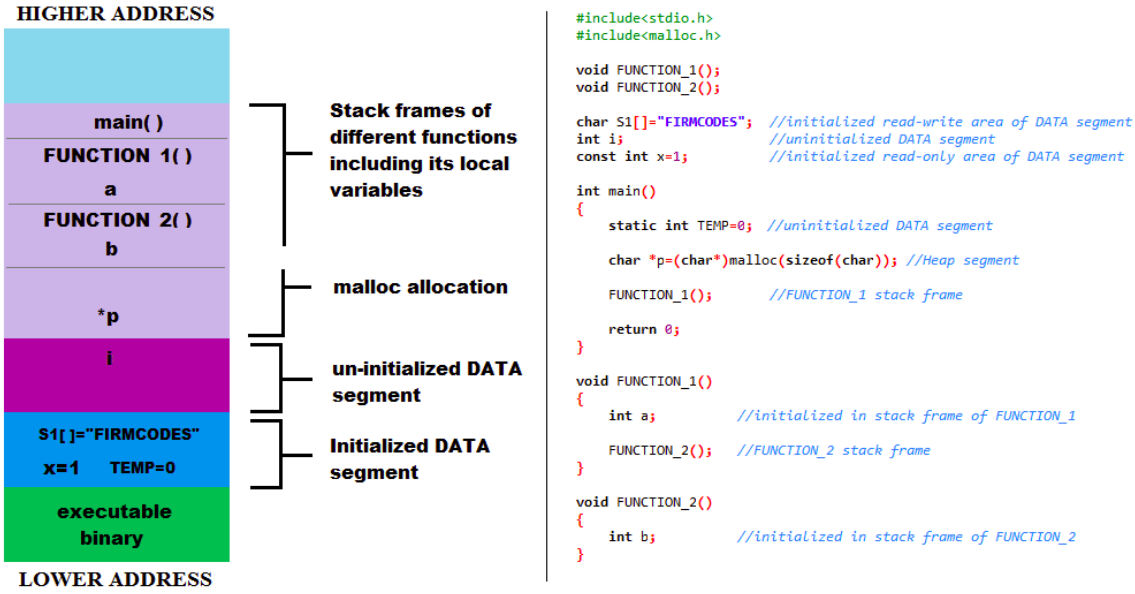
When we call any function, stack frame is created and when function returns, stack frame is destroyed including all local variables of that particular function.

Stack frame contain some data like return address, arguments passed to it, local variables, and any other information needed by the invoked function.

A “stack pointer (SP)” keeps track of stack by each push & pop operation onto it, by adjusted stack pointer to next or previous address.

Unmapped or reserved segment

Unmapped or reserved segment contain command line arguments and other program related data like lower address-higher address of executable image, etc.



As you can see on above figure(click on it for large view), we write a simple code in which we use different variable declaration and three function including main( ) function on right half pic. And on left half of pic, i shown you that how this C program going to store in RAM. Left half of this pic is memroy layout of this C program.

As we discussed in previous tab of Memory layout of C program, your program is stored in memory(RAM) in particular fashion when it is executing and it divides in some blocks. Now we understand those blocks by using our example code seen above on right half of pic.

Text segment:- when we compile our code, we get executable code(which may be in any form like “.bin” , “.exe” and “.hex” , etc) of our C program

inform of output of compilaton. That particular executable code is goes to text segment and would be read only.

Data segment

Initialized Data segment:-

- A string char S1[ ]=”FIRMCODES” and static TEMP=0 will store in initialized read-write area because we don’t use keyword like const or any other which makes variable read-only.
- Keyword static make any variable alive even if its scope is not exist or in simple word, static keyword keeps variable value through out program life.
- Const int x=1 will store in read only area because keyword const make it read only portion and prevent any modification to it.

Uninitialized Data segment:-

- In our program, int i declared global goes to this area of storage because it is not initialized or initialized to zero.

Heap segment:-

- When you compile your program, memory space allocated by other than dynamic allocation (that all locals & global variables) is fixed at compile time, but dynamic memory is not fixed. We can declared and access memory on run time by dynamic allocation techniques that’s why memory allocated by malloc( ),calloc( ) or new is called dynamic memory or run time allocation.
- In example program, we allocate 1 byte dynamic memory using malloc function and stored its address in pointer ‘p’ to keep track of that memory or to access it. We neither freed that memory nor use it. We just want to show you that how that memory actually stored in run time.
- If you freed that memory, the space(here 1 byte) acquired by that memory allocation is not goes back to system and create Heap fragmentation block.

Stack:-

- Any C program contain atleast one main( ) function. So it creates stack frame cause it is also one kind of function. And rest of the all function is called(invoked in proper word) by main( ) function.
- Here we made three function named as main( ), FUNCTION_1( ) and FUNCTION_2( ) which is invoked nested, like main( ) invokes FUNCTION_1( ) and then FUCNTION_1( ) invokes FUNCTION_2( ). All

these function create its own stack frame along with its local variables as you can see on above pic.

- As the FUNCTION_2( ) execution overs its local varible ‘b’ and its stack frame is destroyed, same as FUNCTION_1( ) execution overs its local variable ‘a’ and its stack frame is destroyed, and same thing is done with your main( ) function execution.

Unmapped or reserved segment:-

- As per our point of view, there is nothing in unmapped or reserved region because we dont pass any command line argument to our main.
- But system point of view, unmapped or reserved region contain program information like lower address, higher address and other program related data.

See things practically

```
[root@DEVPC-LINUX-02 vishal]# gcc -o a test.c
[root@DEVPC-LINUX-02 vishal]# size a
   text    data     bss     dec     hex filename
   1061     264      16    1341     53d a
[root@DEVPC-LINUX-02 vishal]#
```

In this figure, we saw the code compilation of our example code(here name test.c), and then use size command to see the size of its executable file(here name ‘a’). And you can see the size of different blocks of text(executable code), data segment, bss, hex , dec segments.

11.Find middle number out of three numbers

```
#include<stdio.h>
int main()
{
    int a = 10,
                b = 15,
                c = 20;
    printf("Middle Number is %d", a>b?  ( c>a? a : (b>c? b:c) )  :  ( c>b? b : (a>c? a:c) )  );
    return 0;
}
```

12.Write a C program to reverse the string without using strrev() ?

Write a C program to reverse the string without using strrev() ?

Write a C program to reverse the string without using strrev() ?

This is simple logic which can reverse whole string without using library function strrev( ) or any other temporary string, we just use one temporary char variable.

```
#include <stdio.h>
#include <conio.h>
#include <string.h>
void main()
{
  char str[10],temp;
  int i,len;
    printf("Enter String : ");
  scanf("%s",str);
    len=strlen(str)-1;
    for(i=0;i<strlen(str)/2;i++)
        {
          temp=str[i];
          str[i]=str[len];
          str[len--]=temp;
        }
  printf("\nReverse string :%s",str);
  getch();
}
```

Probable errors : Here strlen( ) gives you the length of string without null character and we minus 1 from complete length because our for loop will start with 0(zero). If you do not minus length of string by 1 then your printf function will not print anything because you shift last null character on first position and string will end before start.

Reverse string using recursion

```
#include<stdio.h>
#define MAX 10            // Maximum size of string
/* Reverse the String using Recursion */
char* ReverseString(char *str)
{
   static int i=0;
   static char rev[MAX];
   if(*str)
        {
       ReverseString(str+1);
```

```
        rev[i++] = *str;
    }
    return rev;                 // Return index of array every time
}
/* Main method */
int main()
{
        char str[MAX];
        gets(str);
      printf (" %s \n", ReverseString(str));
        return 0;
}
```

13. Write a C program to reverse the words in a sentence in place

```
#include<stdio.h>
/* Driver Functions */
void ReverseString(char *s);
void ReverseWords(char *begin, char *end);
/* Main Method */
int main()
{
        char s[]="This is www.firmcodes.com";
        ReverseString(s);
        printf("%s",s);
        return 0;
}
/* Reverse the string */
void ReverseString(char *s)
{
        char *start_word=s;
        char *traverse=s;

        /* Step 1 : Reverse the individual words */
        while(*traverse)
        {
                if( start_word &&  (*(traverse+1)==' ') || (*(traverse+1)=='\0'))
                {
                        ReverseWords(start_word,traverse);
```

```
                start_word=traverse+2;
            }
            traverse++;
        }

        /* Step 2 : Reverse the whole string */
        ReverseWords(s,traverse-1);
}
/* Reverse individual words */
void ReverseWords(char *begin, char *end)
{
        char temp;
        while(begin<end)
        {
            temp=*begin;
            *begin++=*end;
            *end--=temp;
        }
}
```

14. Write a C program to calculate power(x,n)

```
#include<stdio.h>
unsigned int power(unsigned int x, unsigned int n)
{
   if( n == 0)
      return 1;
   return x*power(x, --n);
}
int main()
{
   unsigned int x = 2;
   unsigned int n = 3;
   printf("%d", power(x, n));
  return 0;
}
```

Write a C program to calculate power(x,n)

Logic is very simple which include recursive call of same function until power become zero and multiply the results as you can see below.

15. Write a c program to find fibonacci series

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

  return 0;
}
```

Fibonacci series program in c using recursion

```
#include<stdio.h>
int Fibonacci(int);
int main()
{
  int n, i = 0, c;
  scanf("%d",&n);
  printf("Fibonacci series\n");
  for ( c = 1 ; c <= n ; c++ )
  {
    printf("%d\n", Fibonacci(i));
```

```
    i++;
  }
  return 0;
}
int Fibonacci(int n)
{
  if ( n == 0 )
    return 0;
  else if ( n == 1 )
    return 1;
  else
    return ( Fibonacci(n-1) + Fibonacci(n-2) );
}
```

16.Write a c Program to convert Decimal to Hexadecimal number

Write a c Program to convert decimal to Hexadecimal number

Write a c Program to convert decimal to Hexadecimal number

Hexadecimal number system: It is base 16 number system which uses the digits from 0 to 9 and A, B, C, D, E, F. Used by computers to easily represent binary numbers.

Decimal number system: It is base 10 number system which uses the digits from 0 to 9. Used by humans.

```
#include<stdio.h>
int main()
{
   long int decimalNumber=17;
        int remainder;
        int quotient;
   int i=1,j;
   char hexadecimalNumber[100];
   quotient = decimalNumber;

   while(quotient!=0)
        {
      remainder = quotient % 16;
    //To convert integer into character
    if( remainder < 10)
```

```
        remainder = remainder + 48;    // Add 48(become ascii later) value to
                                    //remainder if less than 10--see ascii table for more info
    else
      remainder = remainder + 55;      // Add 55(become ascii later) value to
                                  //remainder if greater than 10--see ascii table for more info

    hexadecimalNumber[i++]= remainder;
    quotient = quotient / 16;
   }
   printf("Equivalent hexadecimal value of %d is: ",decimalNumber);
   for(j = i -1 ;j> 0;j--)
    printf("%c",hexadecimalNumber[j]);
  return 0;
}
```

Decimal to hexadecimal conversion algorithm:

Following steps describe how to convert decimal to hexadecimal

Step 1: Divide the original decimal number by 16

Step 2:Adjust the remainder by adding 48 or 55 and store in in char array.

Why 48 for less than 10 and 55 for greater than 10 ?

For example: if decimal is 17 then dividing it by 16, gives 1 as a remainder. Now 1 is less than 10, so we add 48 to 1 and it become 49 which is ASCII value of 1 in as you can see in ASCII table below pic.

Same as if we take example of 12 as a decimal number which is ‘C’ in hex. So remainder become 12 and we add it with 55 which becomes 67 which is ASCII value of ‘C’. See ASCII table below

Step 3: Divide the quotient by 16

Step 4: Repeat the step 2 until we get quotient equal to zero.

Decimal to hexadecimal conversion example:

For example we want to convert decimal number 17 in the hexadecimal.

17/ 16 gives Remainder : 1 , Quotient : 1
1/ 16 gives Remainder : 1 , Quotient : 0

So equivalent hexadecimal number is: 11
That is (17)10 = (11)16

17. Write C code to dynamically allocate one, two and three dimensional arrays (using malloc())

Write C code to dynamically allocate one, two and three dimensional arrays (using malloc())

Write C code to dynamically allocate one, two and three dimensional arrays (using malloc())

Its pretty simple to do this in the C language if you know how to use C pointers. Here are some example C code snipptes….

One dimensional array

```
#include <stdio.h>

#include <stdlib.h>

int main()

{

  int i, count = 0, no_of_elements=5;

        // Allocating dynamic array

  int *array = malloc(no_of_elements * sizeof(int));

        // Accessing one dimensional array

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

      array[i] = ++count;  // OR *(arr+i) = ++count

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

            printf("%d ", array[i]);

        return 0;

}
```

Two dimensional array

There are more than one methods to allocate two dimensional array dynamically but we use only pointer method.

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

```
#define ROW       3
#define COLUMN    4

int main()
{
   int i, j, count=0;
                // Allocating two dimensional dynamic array
   int **arr = (int **)malloc(ROW * sizeof(int *));
   for (i=0; i<ROW; i++)
       arr[i] = (int *)malloc(COLUMN * sizeof(int));
                        // Accessing two dimensional array
   for (i = 0; i <  ROW; i++)
   {
        for (j = 0; j < COLUMN; j++)
        {
                arr[i][j] = ++count;  // OR *(*(arr+i)+j) = ++count
                printf("%d ", arr[i][j]);
        }
   }
    return 0;
}
```

Three dimensional array

```
#include<stdio.h>
#include<malloc.h>
 #define AXIS_X 3
#define AXIS_Y 4
#define AXIS_Z 5
 int main()
{
        int ***p,i,j,k,count=0;
                        // Allocating three dimensional dynamic array
        p=(int ***) malloc(AXIS_X * sizeof(int ***));
        for(i=0;  i<AXIS_X;  i++)
        {
                p[i]=(int **)malloc(AXIS_Y * sizeof(int *));

                for(j=0;  j<AXIS_Y;  j++)
```

```
            {
                p[i][j]=(int *)malloc(AXIS_Z * sizeof(int));
            }
        }

            // Accessing three dimensional array
        for(k=0;  k<AXIS_Z;  k++)
        {
            for(i=0;  i<AXIS_X;  i++)
            {
                for(j=0;  j<AXIS_Y;  j++)
                {
                *(*(*(p+i)+j)+k)= ++count;      // OR *(*(*(p+i)+j)+k) = ++count
                    printf("%d ",p[i][j][k]);
                }
            }
        }
        return 0;
}
```

18. Difference between typedef and #define

Difference between #define and typedef

The #define is a C-directive which is also used to define the aliases for various data types similar to typedef but with following differences:

- The typedef is limited to giving symbolic names to types only where as #define can be used to define alias for values as well, like you can define 1 as ONE etc.
- The typedef interpretation is performed by the compiler where as #define statements are processed by the pre-processor.
- #define should not be terminated with semicolon, but typedef should be terminated with semicolon.

```
typedef char* PTR_t;

#define PTR_d char*
```

- #define will just copy-paste the definition values at the point of use, while typedef is actual definition of a new type

```
typedef char* PTR_t;

#define PTR_d char*
```

PTR_t p1,p2; // In typedef usage, Both p1 and p2 are become char*

PTR_d ptr1,ptr2; // In macro usage, only ptr1 becomes of type char* while ptr2 is char only

- typedef follows the scope rule which mean if a new type is defined in a scope(inside a function), then the new type name will only be visible till the scope is there.
- But in preprocessor case, when preprocessor encounters a #define, then it will replace all the occurrences, after that (No scope rule is followed). See below example:

```
#include <stdio.h>
void print()
{
        #define ONE 1
        typedef int integer;
        printf("%d",ONE);
}
int main()
{
    printf("%d",ONE);      // Print 1 successfully
    integer a;          // Gives compiler error
    return 0;
}
```

- There are certain types of definitions which you can only define using typedef only and not #define like below

typedef unsigned int U_INT32; // We cant define this with #define, because it contain more that one word in definition(unsigned and int here) .

- There is some benefit of typedef when we define structure with typedef like this

```
typedef struct
{
     int x;
     char y;
     float z;
}MyStruct;
//Now define new objects
```

MyStruct A; // You dont need to use struct repeatedly when you define structure again.

Problems on typedef and #define

This problems will clear some doubts and give you practical idea

1.What is the output of this C code?

```
#include <stdio.h>
typedef struct student
{
   char *a;
}stu;
void main()
{
   struct student s;
   s.a = "hey";
   printf("%s", s.a);
}
```

a) Compile time error
b) Varies
c) he
d) hey

2.What is the output of this C code?

```
#include <stdio.h>
typedef struct p
{
   int x, y;
};
int main()
{
   p k1 = {1, 2};
   printf("%d\n", k1.x);
        return 0;
}
```

a) Compile time error
b) 1
c) 0
d) Depends on the standard

3.The correct syntax to use typedef for struct is.

a) typedef struct temp
```
   {
      int a;
   }TEMP;
```
b) typedef struct
```
   {
      int a;
    }TEMP;
```
c) struct temp
```
   {
      int a;
   };
   typedef struct temp TEMP;
```

d) All of the mentioned

4. typedef int (*PFI)(char *, char *)creates
a) type PFI, for pointer to function (of two char * arguments) returning int
b) Error
c) type PFI, function (of two char * arguments) returning int
d) type PFI, for pointer

5. For the following expression to work, which option should be selected.
string p = “HELLO”;
a) typedef char [] string;
b) typedef char *string;
c) Both (a) and (b)
d) Such expression cannot be generated in C

6.What is the output of this C code?

```
#include<stdio.h>
typedef char* charPtr_t;
#define charPtr_d char*
int main()
{
        charPtr_t a,b;
        charPtr_d x,y;
        printf("%d,%d,%d,%d",sizeof(a),sizeof(b),sizeof(x),sizeof(y));
```

```
        return 0;
}
```

a) Compiler error
b) 8,8,8,8
c) 1,1,1,1
d) 8,8,8,1

Answers: (1: d) , (2: a) , (3: d) ,(4: a) , (5: b), (6: d)

19. Difference between macro and inline in c

Difference at Compile Time Expansion

Inline functions are similar to macros because they both are expanded at compile time, but the macros are expanded by the pre-processor, while inline functions are parsed by the compiler.

```
#include<stdio.h>
#define SQAURE(X) X*X
inline int sqaure(int x)
{
        return x*x;
}
int main()
{
        printf(" %d",SQAURE(2+3));
        printf("\n %d",sqaure(2+3));
        return 0;
}
```

In above example, in case of macro expected output would be 25 but it will give you 11 because macros are substituted at compile time so SQAURE(X) would be like this ***#define SQAURE(2+3) 2+3*2+3***
and we know that in c operation priority goes to multiply operator i.e *, so 2*3 gives 6 and then addition of 2+6+3 becomes 11.

While in case of inline function argument is evaluated first and so x would be 5 which gives 5*5 = 25.

Difference at Argument Evaluation

Expressions passed as arguments to inline functions are evaluated once. In some cases, expressions passed as arguments to macros can be evaluated more than once.

```
#include<stdio.h>
#define SQAURE(X) X*X
```

```
inline int sqaure(int x)
{
	return x*x;
}
int main()
{
	int y = 3;
	int x = 3;
	printf(" %d",SQAURE(++x));
	printf("\n %d",sqaure(++y));
	return 0;
}
```

In this example, argument passed to macro will evaluate twice so output of macro function would be like 5 * 5 which gives 25 as a output while in case of inline function argument is evaluated only once so output would be 16.

Difference at Argument type checking

Inline follows strict parameter type checking, macros do not.

```
#include<stdio.h>
#define SQAURE(X) X*X
inline int sqaure(int x)
{
	return x*x;
}
int main()
{
	printf(" %f", SQAURE(1.1));		// Run successfully
	printf("\n %d",sqaure(1.1));	// Gives a compile time error or if run answer will not sure
	return 0;
}
```

This is more specific clear case about argument data type checking, when we pass float argument to macro function it gives output properly i.e 1.21. But when we pass 1.1 to inline function whose argument type is int, it gives either compiler error or 0 as output.

More about Macros and Inline

Macros are always expanded by pre-processor, whereas compiler may or may not replace the inline definitions. You cant force compiler to make a function inline. It is purely compiler based decision.

Debugging macros is also difficult. This is because the preprocessor does the textual replacement for macros, but that textual replacement is not visible in the source code itself. Because of all this, it's generally considered a good idea to use inline functions over macros.

20. Write your own trim() or squeeze() function to remove the spaces from a string

```
#include<stdio.h>
void trim(char *s);
int main()
{
   char str[]=" www . firmcodes . com ";
   trim(str);
   printf("%s",str);
   return 0;
}
void trim(char *s)
{
        char *trimed=s;
        while(*s)     // Check for end of string until null character appear
        {
                *trimed=*s; //shift each character
                if( *(s+1)==' ' )      //Check for space if space appear jump that address
                        ++s;
                if( *(s+1)=='\0' )    //look for null character if null present, terminate
shifting
                        *(trimed+1)='\0';
                trimed++;
                s++;
        }
}
```

21. Write your own C program to implement the atoi() function

The C library function int atoi(const char *str) converts the string argument str to an integer (type int). This problem is about to write own atoi( ) function which has

same argument parameter and return type which also checks for null string and string contain any character.

Declaration

int atoi(const char *str);

Algorithm

Algorithm is pretty simple, we know that char type is hold the ASCII value of any number or character so we **just minus the ASCII value of zero from the given string's individual character** and manage it by multiply by 10 and adding sequentially. See below code for more clarification

```
#include <stdio.h>
int Atoi(char *str)
{
    if (*str == '\0')
      return 0;
    int res = 0;              // Initialize result
    int sign = 1;             // Initialize sign as positive
    int i = 0;                // Initialize index of first digit
     if (str[0] == '-')
    {
        sign = -1;
        i++;                  // Also update index of first digit
    }

    for (; str[i] != '\0'; ++i)
    {
        if ( str[i] <= '0' || str[i] >= '9')      // If string contain character it will terminate
            return 0;
        res = res*10 + str[i] - '0';
    }
    return sign*res;
}
int main()
{
    char str[] = "1234";
    printf("%d ", Atoi(str));
    return 0;
}
```

22. Convert decimal number into hexadecimal octal binary – single universal logic

Convert decimal number into hexadecimal octal binary – single universal logic

Convert decimal number into hexadecimal octal binary – single universal logic

```
#include <stdio.h>
char *convert(unsigned int num, int base) ;
int main( )
{
        printf("%s", convert(16,8));
        return 0;
}
char *convert(unsigned int num, int base)
{
        static char buffer[33];
        char *ptr;
        ptr = &buffer[32];
        *ptr = '\0';
        do
        {
                *--ptr = "0123456789ABCDEF"[num%base];
                num /= base;
        }while(num != 0);
        return(ptr);
}
```

23. Write a Macro's Set,clear and toggle n'th bit using bit wise operators?

```
#include<stdio.h>
void bin(unsigned n);
#define SetBit(number,bit) (number|=(1<<bit-1))
#define ClearBit(number,bit) (number&=(~(1<<bit-1)))
#define Toggle(number,bit) (number ^ (1<<bit-1))
int main()
{
        int i=7;
        printf("\n Binary of 7 is      : ");
        bin(i);

        i=SetBit(i,4);
        printf("\n\n Set 4th bit of 7    : ");
```

```
        bin(i);
        i=ClearBit(i,4);
        printf("\n\n Clear 4th bit of 7  : ");
        bin(i);
        i=Toggle(i,1);
        printf("\n\n Toggle 1st bit of 7 : ");
        bin(i);
        return 0;
}
void bin(unsigned n)
{
   unsigned i;
   for (i = (1 << 31) ; i > 0; i = (i>>1))
      if(n & i)
                        printf("1");
               else
                        printf("0");
}
```

24. Write a C program to convert Little endian to Big endian integer

What is Endianess ?

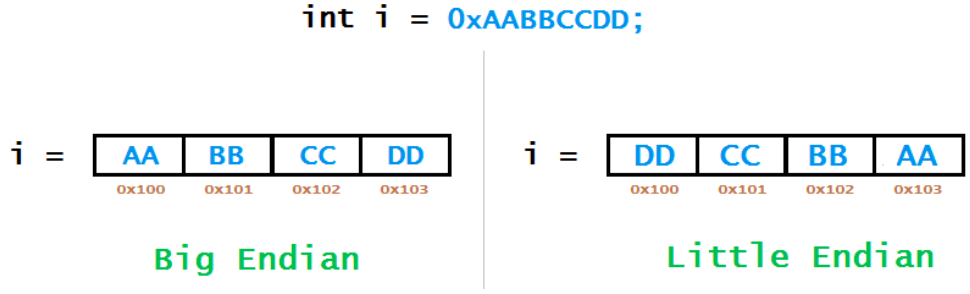
Little and Big Endian are the ways of storing multi-byte data likes int, float, double etc.

Little Endian

In little endian machines, last byte of binary representation of the multi-byte data is stored first.

Big Endian

In big endian machines, first byte of binary representation of the multibyte data is stored first.



Code to convert Little endian integer to Big endian

```
#include <stdio.h>
unsigned int LitToBigEndian(unsigned int x);
int main( )
{
        int Little_Endian = 0xAABBCCDD ;
        printf("\n Little_Endian = 0x%X\n",Little_Endian);
        printf("\n Big_Endian    = 0x%X\n",LitToBigEndian(Little_Endian));
}
unsigned int LitToBigEndian(unsigned int x)
{
        return (((x>>24) & 0x000000ff) | ((x>>8) & 0x0000ff00) | ((x<<8) &
0x00ff0000) | ((x<<24) & 0xff000000));
}
```

25. Write a program to implement XOR functionality with out using XOR(^) operator

If you have your logic cleared with basic AND and OR gates, then this question is piece of cake for you.

Boolean equation for XOR gate is

XOR =AB´+ A´B

In C language, B´ (B 'not') is done by ~ operator. So now logic is pretty easy you can see in below code.

```
#include <stdio.h>
inline int Xor(int x, int y)
{
        return ( (~x) & y ) | ( x & (~y) );
}
int main( )
{
        printf("%d",Xor(1,3));              // Answer would be 2 and it run successful
        return 0;
}
```

26. What's the use of fopen(), fclose(), fprintf(), fscanf(), feof(), fseek(), rewind(), fread(), fwrite(), fgets(), fputs() ?

Here i use the single program to explain almost all file I/O operations simply.

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

```
#include <string.h>
 int main()
{
	FILE *fp;
	char *string="This is testing for fwrite.\n";
	char buff[255];
	int n=0;

	/* File Write Operations */
	fp = fopen("test.txt", "w+");

	/* Access control flags */
	/*
	"r" -> Open for reading.
	"w" -> Open for writing.
	"a" -> Open for appending.
	"r+" -> Both reading and writing.
	"w+" -> Both reading and writing, create new file if it exists,
	"a+" -> Open for both reading and appending.
	*/
	fprintf(fp, "This is testing for fprintf.\n");
	fputs("This is testing for fputs.\n", fp);
	fwrite(string, strlen(string),1,fp);
	fclose(fp);
	memset(buff,0x00,sizeof(buff));
	/* File Reading Operations */
	fp = fopen("test.txt", "r");
	fread(&buff,sizeof(buff),1,fp);
	printf(" fread < Read in Blocks of Array OR Structure > : ");
	printf(" \n\n test.txt \n==============================");
	printf(" \n%s==============================\n", buff );

	rewind(fp); //Reset the cursor position in file at first character
	printf("\n\n fgetc <Fetch single character> : %c\n\n", fgetc(fp) );

	fscanf(fp, "%s", buff);
	printf(" fscanf <Fetch single word starting from current position> : %s\n\n",
buff );
```

```
        fgets(buff, 255, (FILE*)fp);
        printf(" fgets <Fetch whole line from current position> : %s\n", buff );

        /* Counting number of characters in file */
        fseek(fp,0L,0); // Same as rewind.
        /*
        fseek(fp, offset, position);
                Position can be
                0->start of file
                1->current position
                2->end of file
                fseek(fp,0L,1); // Stay at current position.
                fseek(fp,0L,2); // Past end of file.
                fseek(fp,m,0); // Move to (m+1) byte.
                fseek(fp,m,1) // Go forward m bytes.
                fseek(fp,-m,1); // Go backward m bytes from current position.
                fseek(fp,-m,2); // Go backward from end of file.
        */
        while (fgetc(fp) != EOF)
                        ++n;
        if (feof(fp))
                        printf (" feof() < Total characters > : %d\n\n",n);
        fclose(fp);
        return 0;
}
```

27. Sorting Algorithms in C

Sorting Algorithms in C

Sorting Algorithms in C programming is vast topic and often used in most common interview questions to check the logic building aptitude. Sorting in general refers to ordering things based on criteria like numerical, chronological, alphabetical, hierarchical etc.

For example :-Keeping business records and want to sort them by ID number or last name of client? Then you'll need a sorting algorithm.

There are many articles on internet about the different sorting algorithms but i have discussed here some of them basics and inefficient. To understand the more complex and efficient sorting algorithms, it's important to first understand the simpler and slower algorithms.

Fact –It is estimated that around 25% of all CPU cycles are used to sort data.

Selection Sort

Algo – Repeatedly finding the minimum element and swapping.

It basically determines the minimum (or maximum) of the list and swaps it with the element at the index where its supposed to be. The process is repeated such that the nth minimum (or maximum) element is swapped with the element at the n-1th index of the list. The below is an implementation of the algorithm in C.

```
void SelectionSort(int A[], int size)
{
	for(int i=0; i<size-1; i++)
	{
		int Imin = i;
		for(int j=i+1; j<size; j++)
		{
			if( A[j] < A[Imin] )
			{
				Imin = j;
			}
		}
		int temp = A[Imin];
		A[Imin] = A[i];
		A[i] = temp;
	}
}
```

Bubble Sort

Algo – Swapping the adjacent elements Bubble Sort works by comparing each element of the list with the element next to it and swapping them if required. With each pass, the largest of the list is “bubbled” to the end of the list whereas the smaller values sink to the bottom. It is similar to selection sort although not as straight forward. Instead of “selecting” maximum values, they are bubbled to a part of the list. An implementation in C.

```
void BubbleSort(int A[], int size)
{
	for(int i=0; i<size; i++)
```

```
    {
        for(int j=0; j<size-1; j++)
        {
            if( A[j] > A[j+1] )
            {
                int temp = A[j];
                A[j] = A[j+1];
                A[j+1] = temp;
            }
        }
    }
}
```

Insertion sort

Algo – Shift elements one by one Try recalling how you sort a deck of cards. You start from the beginning, traverse through the cards and as you find cards misplaced by precedence you remove them(like making space/here we call it hole/) and insert them back into the right position. Eventually what you have is a sorted deck of cards. The same idea is applied in the Insertion Sort algorithm. The following is an implementation in C.

```
void InsertionSort(int A[], int size)
{
    for(int i=1; i<size; i++)
    {
        int value = A[i];
        int hole = i;
        while( hole>0 && A[hole-1]>value)
        {
            A[hole] = A[hole-1];
            hole--;
```

```
            }
            A[hole] = value ;
        }
}
```

Merge Sort

Algo – Divide and Conquer algorithm or Merge two arrayMerge sort is a recursive algorithm that continually splits a array in equal two halves. If the array is empty or has one item, it is sorted by definition (the base case). If the array has more than one item, we split array and recursively invoke a merge sort on both halves. Once the two halves are sorted, the fundamental operation, called a merge, is performed. Merging is the process of taking two smaller sorted array and combining them together into a single, sorted, new array.

```
void Merge(int A[], int L[], int R[], int L_len, int R_len)
{
    int i=0,j=0,k=0;
    while( i<L_len && j<R_len )
    {
        if( L[i] <= R[j] )
        {
            A[k] = L[i];
            i++;
        }
        else
        {
            A[k] = R[j];
            j++;
        }
        k++;
    }
    while( i<L_len)
    {
        A[k] = L[i];
        i++;k++;
    }
    while( j<R_len )
    {
```

```
            A[k] = R[j];
            j++;k++;
        }
}
void MergeSort(int A[],int size)
{
        if(size<2)return;
        int L_len = size/2;
        int R_len = size - L_len;
        int L[L_len],R[R_len];
        for(int i=0;i<L_len;i++)
        {
            L[i] = A[i];
        }
        for(int j=0;j<R_len;j++)
        {
            R[j] = A[j+L_len];
        }
        MergeSort(L,L_len);
        MergeSort(R,R_len);
        Merge(A,L,R,L_len,R_len);
}
```

Quick Sort

Algo – Divide and Conquer algorithm or Partitioning array

1.) Choose any element as pivot(here i chose last) element from the array list.

2.) Partitioning the array so that all the elements with value less than the pivot will come before the pivot and the element with value greater will come after the pivot with in the same array, which make pivot element in the sorted position.(If the reverse the order we are reversing the sorting order that is descending).

3.) Apply recursively the 3rd step to the sub array of the element with smaller values and separate the sub array of the elements with the greater values until partition of array containing 2 elements.

```
#include <stdio.h>
void swap(int *x, int *y);
int Partition(int A[], int start, int end);
void QuickSort(int A[], int start, int end);
int main()
```

```
{
	int A[] = {3,2,1,5,6,4};
	QuickSort(A,0,5);
	for(int i=0;i<6;i++)
		printf(" %d",A[i]);
	return 0;
}
void swap(int *x, int *y)
{
	int temp = *x;
	*x = *y;
	*y = temp;
}
int Partition(int A[], int start, int end)
{
	int pivot = A[end];
	int pIndex = start;
	for(int i=start; i<end; i++)
	{
		if( A[i] <= pivot )
		{
			swap(&A[i],&A[pIndex]);
			pIndex++;
		}
	}
	swap(&A[end],&A[pIndex]);
	return pIndex;
}
void QuickSort(int A[], int start, int end)
{
	if(start<end)
	{
		int pIndex = Partition(A,start,end);
		QuickSort(A,start,pIndex-1);
		QuickSort(A,pIndex+1,end);
	}
}
```

28. What are practical uses of function pointers in c ?

As we have learn about function pointer from basic to advance level in article Function Pointers in C/C++, now its time to learn What are practical uses of function pointers in c ?

There are many use of function pointer but all they are summing around callback construct, so here i write two use cases for function pointers upon callback construction:

=> **Implement Callback functions –** used for Event Handlers, parser specialization, comparator function passing.

=> **Dynamically function calling(One kind of callback use case) –** Create plugins and extension, Enable-Disables some features upon certain events, creating Finite State Machines(FSM), etc

What is a Callback function?

In simple terms, a Callback function is one that is not called explicitly by the programmer. Instead, there is some mechanism that continually waits for events to occur, and it will call selected functions in response to particular events.

Thus, callbacks allow the user of a function to fine-tune it at runtime(Dynamically function calling).

Practical scenario of Callback

Error Signaling in UNIX – A Unix program, for example, might not want to terminate immediately when it receives SIGTERM, to make sure things get taken care of, it would register the cleanup function as a callback.

Typing certain key combinations at the controlling terminal of a running process in UNIX causes the system to send certain signals:

- **Ctrl+C** sends an INT signal (**SIGINT**); by default, this causes the process to terminate.
- **Ctrl+Z** sends a TSTP signal (**SIGTSTP**); by default, this causes the process to suspend execution.
- **Ctrl+** sends a QUIT signal (**SIGQUIT**); by default, this causes the process to terminate and dump core.

There are more than 30 signals which are generated based on certain situation arise. Here is small example of SIGINT handler to clarify things more.

```
/* signal example */
#include <stdio.h>
```

```
#include <signal.h> /* signal, raise, sig_atomic_t */
void my_handler (int param)
{
	printf (" Signal raised by pressing Ctrl+C.\n");
}
int main ()
{
	void (*prev_handler)(int);
	prev_handler = signal (SIGINT, my_handler);
	while(1)
	{
	//Do some stuff, Wait to signal arise
	}
	return 0;
}
```

Finite State Machines(FSM)

Finite State Machinesm or simply state machine where the elements of (multi-dimensional) arrays indicate the routine that processes/handles the next state.

Enabling features and disabling of features can be done using function pointers. You may have features that you wish to enable or disable that do similar yet distinct things. Instead of populating and cluttering your code with if-else or switch constructs testing variables, you can code it so that it uses a function pointer, and then you can enable/disable features by changing/assigning the function pointer. If you add new variants, you don't have to track down all your if-else or switch cases (and risk missing one); instead you just update your function pointer to enable the new feature, or disable the old one.

Reducing code cluster

```
/* Finite State Machine FSM */
switch(a)
{
	case 0:
			func0();
			break;
	case 1:
			func1();
			break;
	case 2:
			func2();
```

```
                break;
        case 3:
                func3();
                break;
        default:
                funcX();
                break;
}
```

Can be simplified to …

```
/* This declaration may be off a little, but I am after the essence of the idea */
void (*funcArray)(void)[] = {func0, func1, func2, func3, funcX};
... appropriate bounds checking on 'a' ...
funcArray[a]();
```

29. Write a C program to increment number without using + sign

```
#include <stdio.h>
int Increment(unsigned int i)
{
        unsigned int mask = 1;
        while (i & mask)            // Step 1
        {
                i = i & (~mask);    // Step 2
                mask = (mask<<1);       // Step 3
        }
        i = i | mask;                   // Step 4
        return i;
}
int main( )
{
        printf("%d", Increment(1));
        return 0;
}
```

Explanation with example – Let we take example of number 1, so output would be 2 as we know, but we would go through each step separately so that logic become more clear.

Step 1: – Binary of ‘1’ is 0001 which will AND(&) with mask and result would be 1.

Step 2: – this step is simplified like this,

```
        i = i & (~mask);    // Step 2
```

i = (0001) & (~0001);

i = (0001) & (1110);

i = 0

Step 3: – mast now shifted left, so it become 2,

Step 4: – Now value of i = 0, mask = 2

So, i = i | mask; // Step 4

i = 0000 | 00010;

i = 2 , and here is output

30. Write down the program to tell whether the stack is growing in which direction in memory

The code you see here is **compiler dependent** (and also uses some nonstandard or undefined features).

It is completely possible for a compiler to allocate specific stack space for each function and run the allocation backwards to the order in which the variables are declared.

So, in order reliably to determine which way the stack grows, you will need to call a function that uses the address of a variable in the calling function and then determine whether the address of the variable in the called function is less than or more than that of the calling function.

You can probably do it the opposite way by having the called function return the address of the variable in locals, but there are some compilers that deliberately prevent this (for good reason).

To check the direction of stack growth, we just declare two local variable in different scope and compare their address.

```
#include <stdio.h>
void stack(int *local_1)
{
	int local_2;
	printf("\n Address of local_2 : [%u]", &local_2);
	if(local_1 < &local_2)
	{
		printf("\n\n Stack is growing downwards.\n");
	}
	else
	{
```

```
                printf("\n\n Stack is growing upwards.\n");
        }
}
int main( )
{
        int local_1;
        printf("\n Address of local_1 : [%u]", &local_1);
        stack(&local_1);
        return 0;
}
```

31. C program for binary search

C program for binary search

Binary search is generally employed on sorted array. Suppose you have sorted array and you want to search for a single element in array then a simple approach is to scan whole array to find particular element like this

```
// Simple approach
// If x is present then return its location,  otherwise return -1
int search(int arr[], int n, int x)
{
   int i;
   for (i=0; i<n; i++)
      if (arr[i] == x)
       return i;
   return -1;
}
```

Algorithm

The idea of binary search is to use the information that the array is sorted and reduce the time complexity to O(Log n). We basically ignore half of the elements just after one comparison.

1) Compare element to search with the middle element.

2) If element matches with middle element, we return the mid index.

3) Else If element is greater than the mid element, then element can only lie in right half sub array after the mid element. So we recur for right half.

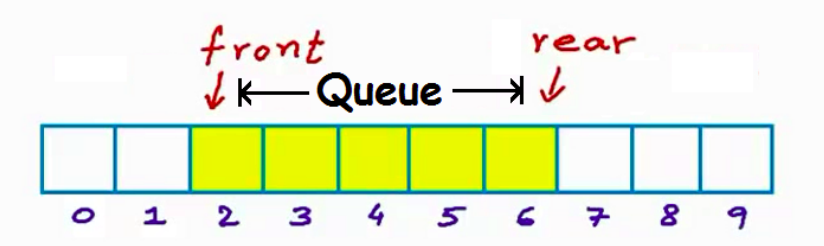
4) Else (element is smaller) recur for the left half.

```
#include<stdlib.h>
#include<stdio.h>
	/* Driver Functions */
int BinarySearch_Iterative(int A[], int size, int element);
int BinarySearch_FirstOccurrence(int A[], int size, int element);
int BinarySearch_Recursive(int A[], int start, int end, int element);
	/* Main Method */
int main()
{
	int A[] = {0,12,6,12,12,18,34,45,55,99};

printf(" BinarySearch_Iterative() 55 at Index = %d \n",
	BinarySearch_Iterative(A,10,55));
printf(" BinarySearch_Recursive() 17 at Index = %d \n",
	BinarySearch_Recursive(A,0,9,34));
printf(" BinarySearch_FirstOccurrence() 12's first occurrence at Index = %d\n",
	BinarySearch_FirstOccurrence(A,9,12));

	return 0;
}
 /* Iterative method */
int BinarySearch_Iterative(int A[], int size, int element)
{
	int start = 0;
	int end = size-1;
	while(start<=end)
	{
		int mid = (start+end)/2;
		if( A[mid] ==  element)
			return mid;   // Check if element is present at mid
		else if( element < A[mid] )
			end = mid-1;	// If element greater, ignore left half
		else
			start = mid+1;// If element is smaller, ignore right half
	}
	return -1;// if we reach here, then element was not present
}
/* Recursive method */
int BinarySearch_Recursive(int A[], int start, int end, int element)
```

```
{
	if(start>end) return -1;// if start become greater than end, then element was not
				present

	int mid = (start+end)/2;// Calculating mid at every recursive call

	if( A[mid] == element )
	return mid;  // Check if element is present at mid

	else if( element < A[mid] )
		BinarySearch_Recursive(A, start, mid-1, element);	// If element
					//greater, ignore left half
	else
		BinarySearch_Recursive(A, mid+1, end, element);	// If element is
				//smaller, ignore right half
}
	/* Binary search to find first or last occurrence of element */
int BinarySearch_FirstOccurrence(int A[], int size, int element)
{
	int result;
	int start = 0;
	int end = size-1;
	while(start<=end)
	{
		int mid = (start+end)/2;
		if( A[mid] ==  element) // Check if element is present at mid
		{
			result = mid;
			end = mid - 1;	// Change this statement "start = mid + 1", for
					//last occurrence
		}
		else if( element < A[mid] )
			end = mid-1;	// If element greater, ignore left half
		else
			start = mid+1;	// If element is smaller, ignore right half
	}
	return result;			// if we reach here, then return element's
occurrence index
}
```

32. Write a c program to implement a queue using array and linked list

Queue is abstract data type in data structure which works as FIFO principle. FIFO means “First in First out”, i.e the element which we have inserted first will be deleted first and the element that we have inserted last will be deleted last. You can have c program to implement queue using array, using stack and using linked list. Two variables are used to implement queue, i.e “rear” and “front”. Insertion will be done at rear side and deletion will be performed at front side. Real-life example of queues are above which will use concept of queue.



Here we use circular array to implement queue which is quite efficient than tradition simple array implementation.

Array implementation of Queue

```
#include <stdio.h>
 int A[10], front=-1, rear=-1; // Creating Queue
int size_A = sizeof(A)/sizeof(A[0]); // Calculating Size of Array Dynamically
 /* Driver Function */
int IsEmpty();
void Insert(int data);
int Delete();
void PrintQueue();
 /* Main Method */
int main()
{
        Insert(0);
        Insert(1);
        Insert(2);
        Insert(3);
        Delete(); // Delete 0 from starting
```

```
        Delete(); // Delete 1 from starting
        PrintQueue(); //Print Queue
        return 0;
}
 int IsEmpty()
{
        if(rear==-1 && front==-1)
                return 1;
        else
                return 0;
}
 void Insert(int data)
{
        if(IsEmpty())
                front = rear = 0;
        else if( rear+1 == front )
                printf("\n Queue is Full \n");
        else
                rear = (rear+1)%size_A;
        A[rear] = data;
}
 int Delete()
{
        if(IsEmpty())
                printf("\n Queue is Empty \n");
        else
                front = (front+1)%size_A;
        return A[front-1];
}
 void PrintQueue()
{
        int i=0;
        for(i=front;i<=rear;i++)
                printf(" %d",A[i]);
}
```

Linked List implementation of Queue

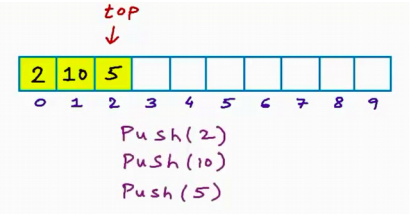
```
#include <stdio.h>
#include <stdlib.h>
/* List Structure */
```

```
typedef struct Node
{
	int data;
	struct Node *link;
}node;
node *head=NULL;		// Head node to keep track of list
/* Driver Functions */
void EnQueque(int data);
int DeQueue();
void print(node *p);
/* Main Method */
int main()
{
	EnQueque(0);
	EnQueque(1);
	EnQueque(2);
	EnQueque(3);
	DeQueue();		// Delete 0 from list
	DeQueue();		// Delete 1 from list
	EnQueque(4);		// Add 4 at the end of list
	print(head);	// Print element of queue
	return 0;
}
/* Insert Element */
void EnQueque(int data)
{
	// Declaring node
	node *temp = (node*)calloc(1,sizeof(node));
	temp->data = data;
	temp->link = NULL;
	// If head is NULL or first node
	if(!head)
	{
		head = temp;
		return;
	}
	node *traverse=head;
	// Traverse list upto end
	while(traverse->link)
```

```
                traverse = traverse->link;
        traverse->link = temp;
}
/* Delete Element */
int DeQueue()
{
        node* temp = head;
        head = head->link;
        int data = temp->data;
        free(temp);
        return data;
}
/* Print queue */
void print(node *p)
{
        printf(" %d",p->data);
        if(!p->link)
                return;
        print(p->link);
}
```

33.Write a c program to implement a stack using an array and linked list

Stack is abstract data type in data structure which works as LIFO principle. LIFO means “Last in First out”, i.e the element which we have inserted last will be used first and the element that we have inserted first will be used last. You can have c program to implement stack using array and using linked list. Single variables are used to implement stack, i.e “top”. Insertion and deletion will be performed at top side. Real-life example of stacks are above which will use concept of stack.




```
#include <stdio.h>
int A[10], top=-1;          // Creating Stack
```

```
int size_A = sizeof(A)/sizeof(A[0]);    // Calculating Size of Array
/* Driver Functions */
void Push(int data);
int Pop();
void PrintStack();
int IsEmpty();
/* Main Method */
int main()
{
        Push(0);
        Push(1);
        Push(2);
        Push(3);
        Pop();                  // Delete 3 from starting
        Pop();                  // Delete 2 from starting
PrintStack();         //Print Stack
return 0;
}
/* Check for empty stack */
int IsEmpty()
{
        if(top==-1)
                return 1;
        else
                return 0;
}

/* Insert Element */
void Push(int data)
{
        if(IsEmpty())
                top = 0;
        else if( top == size_A )
                printf("\n Stack is Full \n");
        else
                top++;

        A[top] = data;
}
```

```
/* Delete Element */
int Pop()
{
	if(IsEmpty())
		printf("\n Stack is Empty \n");
	else
		top--;

	return A[top-1];
}
/* Print Stack */
void PrintStack()
{
	int i=0;
	for(i=0;i<=top;i++)
		printf(" %d",A[i]);
}
```

Stack implementation using Linked List

```
#include <stdio.h>
#include <stdlib.h>
/* List Structure */
typedef struct Node
{
	int data;
	struct Node *link;
}node;
int NoOfNodes=0;		// Count number of node
node *head=NULL;		// Head node to keep track of list
	/* Driver Functions */
void Push(int data);
int Pop();
void PrintStack(node *);
	/* Main Method */
int main()
{
	Push(0);
	Push(1);
	Push(2);
	Push(3);
```

```
	Pop();		// Delete 3 from Stack
	Pop();		// Delete 2 from Stack

	PrintStack(head); // Print Stack data
	return 0;
}
/* Insert Element */
void Push(int data)
{
	// Declaring node
	NoOfNodes++;
	node *temp = (node*)calloc(1,sizeof(node));
	temp->data = data;
	temp->link = NULL;

	// If head is NULL or first node
	if(!head)
	{
		head = temp;
		return;
	}

	// Traverse list upto end
	node *traverse=head;
	while(traverse->link)
		traverse = traverse->link;
	traverse->link = temp;
}
/* Delete Element */
int Pop()
{
	node *traverse = head;
	for(int i=0;i<NoOfNodes-2;i++)
		traverse = traverse->link;

	node *Delete = traverse->link;
	traverse->link = NULL;
	int data = Delete->data;
```

```
        free(Delete);
        NoOfNodes--;

        return data;
}

/* Print Stack */
void PrintStack(node *p)
{
        printf(" %d",p->data);
        if(!p->link)return;
        PrintStack(p->link);
}
```

34. C Program to Print all the Repeated Numbers with Frequency in Array

Write a C Program to Print all the Repeated Numbers with Frequency in Array. So first approach is to use linear search and find the occurrence of numbers but efficient solution is to use less time and space complexity.
For Example if you given array A[ ] = { 5, 3, 2, 5, 3, 1}; to find frequency of numbers, So here is small solution with Time and Space complexity with O(n) .

```
/* Solution in two line  */
for(int i=0; i<SIZE; i++)
        OccurCount[A[i]]++;
```

Here array **A** is our main array in which we want to find frequency of numbers and array **OccurCount** is used to count element occurrences.

Logic

The logic of this problem is pretty simple

1. We declare new array **OccurCount** of same size as A.
2. Increment the value in **OccurCount** array at index same as element. For example if we get first element '5' in array **A** , then increment value of element in array **OccurCount** at index 5**.**

```
#include <stdio.h>
#define SIZE 6
int main()
{
        int A[SIZE] = {1,2,1,4,1,4}; // Main array to search occurrence
```

```
        int OccurCount[SIZE]={0,0,0,0,0,0};  // make zero each element otherwise you
                                             //will see garbage value in output
        /* Solution in two line  */
        for(int i=0; i<SIZE; i++)
                OccurCount[A[i]]++;
        /* Print element occurrences */
        for(int i=0; i<SIZE; i++)
                printf(" %d is occur %d times \n",i,OccurCount[i]);

        return 0;
}
```

35. C program to find the middle node of a linked list

```
/* Function to find the middle of the linked list */
void findMiddle(node *head)
{
   node *slow_ptr = head;
   node *fast_ptr = head;
   if (head!=NULL)
   {
      while (fast_ptr != NULL && fast_ptr->link != NULL)
      {
         fast_ptr = fast_ptr->link->link;
         slow_ptr = slow_ptr->link;
      }
      printf(" The middle element is %d \n", slow_ptr->data);
   }
}
```

But there is another way which is very efficient. Traverse linked list using two pointers named as fast and slow. Move slow pointer by one node and fast pointer by two node in single iteration. When the fast pointer reaches end slow pointer will reach middle of the linked list.

```
#include<iostream>
#include<stdlib.h>
using namespace std;
/* List Structure */
typedef struct Node
```

```
{
        int data;
        struct Node *link;
}node;
node *head = NULL;      // Head node to keep track of linked list
/* Driver functions */
void findMiddle(node *head);
void insert(int data, int position);
void print();

/* Main method */
int main()
{
        insert(0,1);   // Insert Element at first position LINKED-LIST =  / 0 /
        insert(1,2);   // Insert Element at second position LINKED-LIST = / 0 1 /
        insert(2,3);   // Insert Element at third position LINKED-LIST =  / 0 1 2 /
        insert(3,4);   // Insert Element at fourth position LINKED-LIST = / 0 1 2 3/
        insert(4,5);   // Insert Element at fourth position LINKED-LIST = / 0 1 2 3 4/

        print();                // Printing Elements of Linked List
        findMiddle(head); // Find middle element
return 0;
}

/* Function to find the middle of the linked list */
void findMiddle(node *head)
{
   node *slow_ptr = head;
   node *fast_ptr = head;
   if (head!=NULL)
   {
      while (fast_ptr != NULL && fast_ptr->link != NULL)
      {
         fast_ptr = fast_ptr->link->link;
         slow_ptr = slow_ptr->link;
      }
      printf(" The middle element is %d \n", slow_ptr->data);
   }
}
```

```
/* Function for Inserting nodes at defined position */
void insert(int data, int position)
{
	/* Declaring node */
	node *temp = (node*)malloc(sizeof(node));
	temp->data = data;
	temp->link = NULL;

	/* if node insertion at first point */
	if(position==1)
	{
		temp->link = head;
		head = temp;
		return ;
	}

	/* Adding & Adjusting node links*/
	node *traverse = head;
	for(int i=0; i<position-2; i++)
	{
		traverse = traverse->link;
	}
	temp->link = traverse->link;
	traverse->link = temp;
}

/* Function for Printing Linked List */
void print()
{
	printf("\n Linked List = ");
	node *p = head;
	while(p)
	{
		printf(" %d",p->data);
		p = p->link;
	}
	printf(" \n\n");
}
```

36.Write C code to implement the strstr() function to search for a substring

Synopsis:

/* Prototype */
#include <stdio.h>

char *strstr(char *string2, char *string1);

Description:

The strstr function locates the first occurrence of the string string1 in the string string2 and returns a pointer to the beginning of the first occurrence.

Return Value

The strstr function returns a pointer within string2 that points to a string identical to string1. If no such sub string exists in source string a null pointer is returned.

```
#include<iostream>
using namespace std;

/* Driver Function */
char* StrStr(char *str, char *substr)
{
        while (*str)
        {
                char *Begin = str;
                char *pattern = substr;

                // If first character of sub string match, check for whole string
                while (*str && *pattern && *str == *pattern)
                    {
                        str++;
                        pattern++;
                }
                // If complete sub string match, return starting address
                if (!*pattern)
                      return Begin;

                str = Begin + 1; // Increament main string
```

```
        }
        return NULL;
}

/* Main Method */
int main()
{
        char s1 [] = "This is www.firmcodes.com";
   printf ("Returned String 1: %s\n", StrStr(s1, "firmcodes"));
        return 0;
}
```

37. Delete node in linked list without head pointer and traversing

Problem – You have given only pointer to a particular node to delete that node without using head node and traversing

Solution – There is no practical solution possible to delete node directly by given pointer, but there is a way to delete that node logically rather than physically in memory. And the solution is to copy the data from the next node to the current node by given pointer to be deleted and delete the next node. Something like following example

Note: This solution doesn't work if the node to be deleted is the last node of the list.

```
#include<iostream>
#include<stdlib.h>
using namespace std;
/* List Structure */
typedef struct Node
{
        int data;
        struct Node *link;
}node;

node *head = NULL;      // Head node to keep track of linked list

/* Driver functions */
void DeleteNode(node *node_ptr);
void insert(int data, int position);
void print();
```

```
/* Main method */
int main()
{
        insert(0,1);   // Insert Element at first position LINKED-LIST =  / 0 /
        insert(1,2);   // Insert Element at second position LINKED-LIST = / 0 1 /
        insert(2,3);   // Insert Element at third position LINKED-LIST =  / 0 1 2 /
        insert(3,4);   // Insert Element at fourth position LINKED-LIST = / 0 1 2 3/
        insert(4,5);   // Insert Element at fourth position LINKED-LIST = / 0 1 2 3 4/

        print();        // Printing Elements of Linked List

        DeleteNode(head);          //Deleting first node

        print();        // Printing Elements of Linked List

        return 0;
}

/* Delete node by given pointer */
void DeleteNode(node *node_ptr)
{
  node *next = node_ptr->link;
  node_ptr->data = next->data;
  node_ptr->link = next->link;

  free(next);
}

/* Function for Inserting nodes at defined position */
void insert(int data, int position)
{
        /* Declaring node */
        node *temp = (node*)malloc(sizeof(node));
        temp->data = data;
        temp->link = NULL;

        /* if node insertion at first point */
        if(position==1)
```

```
        {
                temp->link = head;
                head = temp;
                return ;
        }

        /* Adding & Adjusting node links*/
        node *traverse = head;
        for(int i=0; i<position-2; i++)
        {
                traverse = traverse->link;
        }
        temp->link = traverse->link;
        traverse->link = temp;
}

/* Function for Printing Linked List */
void print()
{
        printf("\n Linked List = ");
        node *p = head;

        while(p)
        {
                printf(" %d",p->data);
                p = p->link;
        }
        printf(" \n\n");
}
```

38. C program to sorting a singly linked list

```
#include<iostream>
#include<stdlib.h>
using namespace std;

/* List Structure */
typedef struct Node
{
```

```
        int data;
        struct Node *link;
}node;

node *head = NULL;      // Head node to keep track of linked list

/* Driver functions */
void DeleteNode(node *node_ptr);
void insert(int data, int position);
void print();

/* Main method */
int main()
{
        insert(0,1);  // Insert Element at first position LINKED-LIST =  / 0 /
        insert(1,2);  // Insert Element at second position LINKED-LIST = / 0 1 /
        insert(2,3);  // Insert Element at third position LINKED-LIST =  / 0 1 2 /
        insert(3,4);  // Insert Element at fourth position LINKED-LIST = / 0 1 2 3/
        insert(4,5);  // Insert Element at fourth position LINKED-LIST = / 0 1 2 3 4/

        print();        // Printing Elements of Linked List

        DeleteNode(head);         //Deleting first node

        print();        // Printing Elements of Linked List

        return 0;
}

/* Delete node by given pointer */
void DeleteNode(node *node_ptr)
{
  node *next = node_ptr->link;
  node_ptr->data = next->data;
  node_ptr->link = next->link;

  free(next);
}
```

```
/* Function for Inserting nodes at defined position */
void insert(int data, int position)
{
	/* Declaring node */
	node *temp = (node*)malloc(sizeof(node));
	temp->data = data;
	temp->link = NULL;

	/* if node insertion at first point */
	if(position==1)
	{
		temp->link = head;
		head = temp;
		return ;
	}

	/* Adding & Adjusting node links*/
	node *traverse = head;
	for(int i=0; i<position-2; i++)
	{
		traverse = traverse->link;
	}
	temp->link = traverse->link;
	traverse->link = temp;
}

/* Function for Printing Linked List */
void print()
{
	printf("\n Linked List = ");
	node *p = head;

	while(p)
	{
		printf(" %d",p->data);
		p = p->link;
	}
	printf(" \n\n");
}
```

39.Difference between const char *p, char const *p and char *const p

Difference between **const char *p, char const *p** and **char *const p** is most confusing interview question ever faced by c developer or candidate including me. So i have find the answer of this question some how and want to publish as a article to share my knowledge.

Answer:-

First two **const char *p** and **char const *p** both are same i.e. points to constant character(You can change where p points, but you can't change pointed characters using that pointer).

But **char * const p** is a constant pointer to a variable(i.e. you can't change the pointer).

There is also **const char * const p** which is a constant pointer to a constant char (so nothing about it can be changed).

This answer is enough to find difference between these three statements, but not sufficient to build your fundamental concept until you used it practically.

So, let we understand it with example

1). const char *p (Points to constant character)

```
#include<stdio.h>
int main()
{
        char z = 'A';
        char y;
        const char *p = &y; // Can't change pointed characters using pointer, can change pointer
        p = &z;
        /* Invalid, gives compier error "assignment of read-only location *p" */
        // *p = 'B';    // You can't change pointed characters using pointer
        printf(" %c \n",*p);
        return 0;
}
```

2).char * const p (Constant pointer to variable)

```
#include<stdio.h>
 int main()
{
        char z;
        char y;
                /* Invalid, gives compier error "uninitialized const p" */
                // char * const p;    // Need to initialize at declaration time
        char * const p = &z;
                // Address cannot be changed, Must be initialize at declaration time
   *p = 'A';
        /* Invalid, gives compier error "assignment of read-only variable p" */
        // p = &y;     // const pointer can't be change, must be initialize at declaration
time
        printf(" %c \n",*p);
        return 0;
}
```

3). const char * const p (Constant pointer to a constant char)

```
#include<stdio.h>
int main()
{
        char z = 'A';
        char y;
        const char * const p = &z;        // Both address and value cannot be changed,
Initialized at declaration time

        /* Invalid, gives compier error "assignment of read-only variable p" */
        // p = &y;  // Can't change pointer, Need to initialize at declaration time
        /* Invalid, gives compier error "assignment of read-only location *(const
char*)p" */
        // *p = 'B';    // Can’t change pointed characters
        printf(" %c \n",*p);
        return 0;
}
```

Summary

const char *p = Can’t change pointed characters using pointer, can change pointer

char const *p = Can’t change pointed characters using pointer, can change pointer

char * const p = Address cannot be changed, Must be initialize at declaration time

const char * const p = Both address and value cannot be changed, Initialized at declaration time

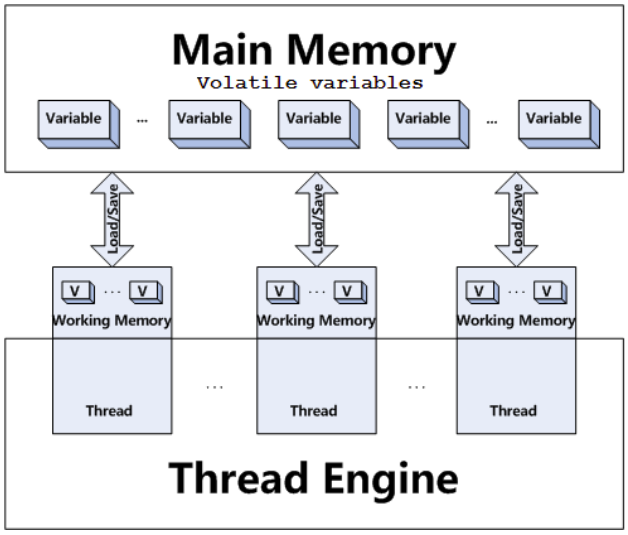
Trick or Tips

Another thumb rule is to check where **const** is:

before * => Can't change pointed characters using pointer

after *=> Can't change pointer stored address

40. Volatile keyword in c language and embedded system



I think, you might have heard of Volatile keyword in c and embedded system frequently. But, when it comes to writing code especially in Embedded programming the use of "volatile" keyword is very often.

But now the question is,

- **What is it (volatile)?**
- **What it does?**
- **Why do we need it?**
- **Does my program/code really need this keyword?**

All these and more I will cover in this article. Go through all below tabs step by step

What does volatile mean?

Ok, let's find out the meaning of volatile in dictionary.

VOLATILE means: – UNSTABLE, UNPREDICTABLE…etc.

So, the basic meaning of volatile is we can't predict what is going to happen next.

What is the significance of volatile keyword?

Volatile keyword is used to inform the compiler not to predict/assume/believe/presume the value of the particular variable which has been declared as volatile.

What is code optimization?

I think this is the particular section is missing everywhere. So now we will see, what is code optimization. Suppose, we have two code segment like below

Code 1:

```
/* Optimization code snippet 1 */
#include<stdio.h>
int x = 0;
int main()
{

	if (x == 0) // This condition is always true
	{
		printf(" x = 0 \n");
	}
	else		// Else part will be optimized because x will never other than 0
	{
		printf(" x != 0 \n");
	}
	return 0;
}
```

code 2:-

```
/* Optimization code snippet 2 */
#include<stdio.h>
volatile int x;		/* volatile Keyword*/
int main()
{
	x = 0;
	if (x == 0)
	{
	printf(" x = 0 \n");
	}
```

```
    else            // Now compiler never optimize else part because the variable is
declared as volatile
    {
    printf(" x != 0 \n");
    }
    return 0;
}
```

In code 1, compiler will optimize the code it will ignore the else part in final executable, because the variable “x” will never ever become other than 0.

In code 2, the compiler will never optimized the code (else part) because, by declaring x as volatile compiler comes to know that this variable can change at any point of time. So compiler does not ignore the else part and consider it in final executable.

Why/When do we need volatile ?

In following case we need to use volatile variable.

- Memory-mapped peripheral registers
- Global variables modified by an interrupt service routine
- Global variables within a multi-threaded application

If we do not use volatile qualifier the following problems may arise:

- Code that works fine-until you turn optimization on
- Code that works fine-as long as interrupts are disabled
- Flaky hardware drivers
- Tasks that work fine in isolation-yet crash when another task is enabled

e.g

```
static int var;
void test( )
{
    var = 1;
    while (var != 10)
        continue;
}
```

The above code sets the value in var to 1. It then starts to poll that value in a loop until the value of var becomes 10.

An optimizing compiler will notice that no other code can possibly change the value stored in ‘var’, and therefore assume that it will remain equal to 0 at all times. The compiler will then replace the function body with an infinite loop, similar to this:

```
void test_opt(void)
{
        var = 0;
        while (1)
                continue;
}
```

How do we declare a volatile variable?

Include the keyword volatile before or after the data type in the variable.

```
/* Declaring volatile variable */
volatile int var;
or
int volatile var;
/* Above both statement are same */
```

Pointer to a volatile variable

```
/* Pointer to a volatile variable */
volatile int * var;
int volatile * var;
/* Above statements implicate ‘var’ is a pointer to a volatile integer */
```

Volatile pointers to non-volatile variables

```
/* Volatile pointers to non-volatile variables */
int * volatile var;
/* Here var is a volatile pointer to a non-volatile variable/object */
```

This type of pointer are very rarely used in embedded programming.

Volatile pointers to volatile variables

```
/* Volatile pointers to volatile variables */
int volatile * volatile var;
```

If we qualify a struct or union with a volatile qualifier, then the entire contents of the struct/union becomes volatile. We can also apply the volatile qualifier to the individual members of the struct/union.

Does my program/code really need this keyword ?

As we have already seen above, when do we need volatile keyword, then next tab consist some cases which really makes you understand.

Usages of volatile qualifier

1). Peripheral registers

Most embedded systems consist of a handful of peripherals devices. The value of the registers of these peripheral devices may change asynchronously. Let say there

is an 8-bit status register at address 0x1234 in any hypothetical device. What we need to do is to poll this status register until it becomes non-zero. The following code snippet is an incorrect implementation of this scenario/requirement:

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

Now, no code in proximity attempts to change the value in the register whose address(0x1234) is kept in the ‘ptr’ pointer. A typical optimizing compiler(if optimization is turned ON) will optimize the above code as below:

```
mov ptr, #0x1234 -> move address 0x1234 to ptr

mov a, @ptr -> move whatever stored at ‘ptr’ to accumulator

loop bz loop -> go into infinite loop
```

What the assumes while optimizing the code is easy to interpret. It simply takes the value stored at the address location 0x1234(which is stored in ‘ptr’) into accumulator and it never updates this value as because apparently the value at the address 0x1234 never gets changed(by any nearby code). So, as the code suggests, the compiler replaces it with an infinite loop (comparing the initial zero value stored at the address 0x1234 with a constant ‘zero’). As the value stored at this address would initially be zero and it is never updated, this loop goes forever. The code beyond this point would never get executed and the system would go into a blank state.

So what we essentially need to do here is to force the compiler to update the value stored at the address 0x1234 whenever it does the comparison operation. The volatile qualifier does the trick for us. Look at the code snippet below:

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

2). ISR(Interrupt Service Routine)

Sometimes we check a global variable in the main code and the variable is only changed by the interrupt service routine. Lets say a serial port interrupt tests 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 serial port ISR sets a particular variable, say ‘etx_rcvd’. And from the main code somewhere else this ‘etx_rcvd’ is checked in a loop and untill it becomes TRUE the code waits at this loop. Now lets check the code snippet below:

```
int etx_rcvd = FALSE;
```

```
int main()
{
	// Do something
	while (!ext_rcvd)
	{
		// Wait
	}
	// Do something
	return 0;
}

interrupt void rx_isr(void)
{
	// Do something
	if (ETX == rx_char)
	{
		etx_rcvd = TRUE;
	}
	// Do something
}
```

This code may work with optimization turned off. But almost all the optimizing compiler would optimize this code to something which is not intended here. Because the compiler doesn't even have any hint that etx_rcvd can be changed outside the code somewhere(as we saw within the serial port ISR). So the compiler assumes the expression !ext_rcvd would always be true and would replace the code with infinite loop. Consequently the system would never be able to exit the while loop. All the code after the while loop may even be removed by the optimizer or never be reached by the program. Some compiler may throw a warning, or some may not, depends completely on the particular compiler.

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, often tasks/threads involved in a multi-threaded application communicate via a shared memory location i.e. through a global variable. Well, a compiler does not have any idea about preemptive scheduling or to say, context switching or whatsoever. So this is sort of same problem as we discussed in the case of an interrupt service routine changing the peripheral memory register. Embedded Systems Programmer has to take care that all shared global variables in an multi-threaded environment be declared volatile. For example:

```
int cntr;
void task1(void)
{
        cntr = 0;
        while (cntr == 0)
        {
        sleep(1);
        }
        // Do something
}
void task2(void)
{
        // Do something
        cntr++;

        sleep(10);
        // Do something
}
```

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.

Some compilers allow you 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.

Can you have constant volatile variable?

You can have a constant pointer to a volatile variable but not a constant volatile variable.

Consider the following two blocks of a program, where second block is the same as first but with volatile keyword.

```
/* Code snippest 1 */
{
        BOOL flag = TRUE;
        while( flag );
        repeat:
        jmp repeat
}
/* Code snippest 2 */
{
        volatile BOOL flag = TRUE;
```

```
        mov dword ptr [flag], 1
        while( flag );
        repeat:
        mov eax, dword ptr [flag]
        test eax, eax
        jne repeat
}
```

In first block variable '**flag**' could be cached by compiler into a CPU register, because it does not have volatile qualifier. Because no one will change value at a register, program will hang in an infinite loop (yes, all code below this block is unreachable code, and compiler such as Microsoft Visual C++ knows about it). Also this loop was optimized in equivalent program with the same infinite loop, but without involving variable initialization and fetching. '**jmp label**' means the same as '**goto label**' in C code.

Second block have volatile qualifier and have more complex assembler output (initializing '**flag**' with '**mov**' instruction, in a loop fetching this flag into CPU register '**eax**' with a '**mov**' instruction, comparing fetched value with zero with '**test**' instruction, and returning to the beginning of the loop if '**flag**' was not equal to zero. '**jne**' means goto if not equal). This is all because volatile keyword prohibits compiler to cache variable value into CPU register, and it is fetched in all loop iterations. Such code is not always is an infinite loop, because another thread in the same program potentially could change value of variable 'flag' and first thread will exit the loop.

It is important to understand that volatile keyword is just a directive for compiler and it works only at a compile-time.

For example, the fact of using interlocked operation differs from just a compiler option, since special assembler commands are produced. Thus, interlocked instructions are most like to hardware directives, and they work at a run-time.

Can a variable be both Volatile and Const ?

- **const** means the program can not modify the value
- **volatile** means the value may be arbitrarily modified outside the program.

The two are separate and not mutually exclusive. Use them together, for instance, in the case of reading a hardware status register. const prevents the value from being stomped on before compilation, while volatile tells the compiler that this value can be changed at any time external to the program.

This,

const volatile <type> <variable name>

will thus satisfy both requirements and prevent an optimizing compiler from incorrectly optimizing the code, that it would do if only “const” were used.

Conclusion

1. A volatile variable can be changed by the background routine of pre-processor. This background routine may be interrupt signals by microprocessor, threads, real times clocks etc.

2. In simple word, we can say a value volatile variable which has stored in the memory can be by any external sources.

3. Whenever compiler encounter any reference of volatile variable is always load the value of variable from memory so that if any external source has modified the value in the memory compiler will get its updated value.

4. Working principle of volatile variable is opposite to the register variable in c. Hence volatile variables take more execution time than non-volatile variables.

I hope this article might have helped you in understanding volatile keyword.
C Program to convert Binary number to Decimal ?

This C Program Converts the given Binary number to Decimal. Binary number is a number that can be represented using only two numeric symbols – 0s and 1s so which is a number in base 2. Decimal is base 10 arithmetic where each digit is a value from 0 to 9.

```
#include <stdio.h>

/* Driver function */
int BinaryToDecimal(long int binary)
{
        long int decimal = 0, i = 1, remainder;

        /* Iterate unitl number becomes zero */
        while (binary != 0)
   {
      remainder = binary % 10;
      decimal = decimal + remainder * i;
      i = i * 2;
      binary = binary / 10;
   }
   return decimal;
```

```
}
/* Main Method */
int main()
{
   long int binary = 10011011;

   printf("Equivalent Decimal value:     %ld \n\n", BinaryToDecimal(binary));
   printf("Equivalent hexadecimal value: %lX \n\n", BinaryToDecimal(binary));
        return 0;
}
```

41. C Program to Display its own Source Code as its Output

Here is source code of the C Program to display its own source code as its output. We have used low-level file handling to achieve this.

```
/* C Program to Display its own Source Code as its Output */
#include <stdio.h>

int main()
{
   FILE *fp;
   char ch;
   fp = fopen(__FILE__,"r");
   while (ch != EOF)
   {
      ch = getc(fp);
      putchar(ch);
   }
        fclose(fp);
   return 0;
}
```

42. structure padding and packing in c example

Data structure alignment is the way data is arranged and accessed in computer memory. It consists of two separate but related issues: **data alignment and data structure padding**. When a modern computer reads from or writes to a memory address, it will do this in word sized chunks (e.g. 4 byte chunks on a 32-bit system) or larger. Data alignment means putting the data at a memory address equal to some multiple of the word size, which increases the system's performance due to the way the CPU handles memory. To align the data, it may be necessary to insert some

meaningless bytes between the end of the last data structure and the start of the next, which is **data structure padding.**

What is 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.

What is Packing

Packing, on the other hand prevents compiler from doing padding means remove the unallocated space allocated by structure.

- In case of Linux we use **__attribute__((__packed__))** to pack structure.
- In case of Windows (specially in Dev c++) use **# pragma pack (1)** to pack structure.

Example:-

1. In case of Padding of Structure

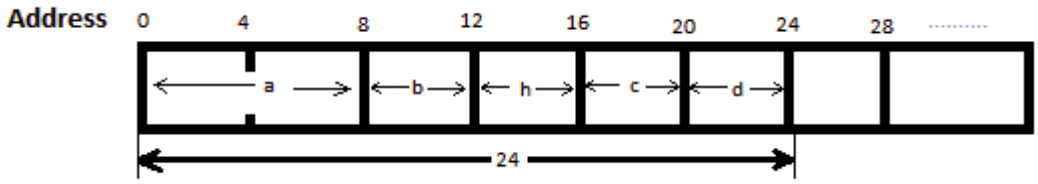
Program

```
#include <iostream>
struct school
{
        double a;
        int b;
        char h;
        int c;
        char d;
};
```

```
int main()
{
        school students;
        printf("size of struct %d \n",sizeof(students));
        return 0;

}
```

Memory Level Description

Size of Structure is 24 .Due to Padding all char variable are also assigned 4 bytes.



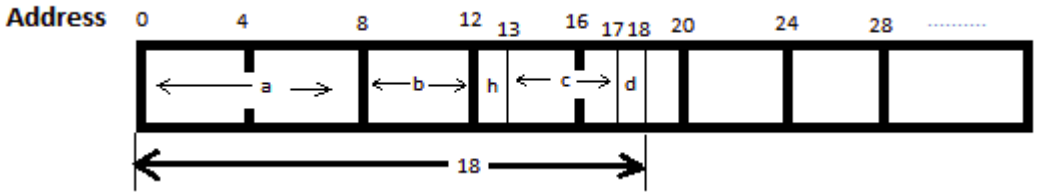
2. In case of Packing of Structure

Program

```
#include <iostream>
# pragma pack (1)
struct school
{
        double a;
        int b;
        char h;
        int c;
        char d;
};

int main()
{
        school students;
        printf("size of struct %d \n",sizeof(students));
        return 0;
}
```

Memory Level Description

Size of Structure is 18. When padding is removed then memory is assigned according to the size of variable.



43. C Program to find whether a number is even or add without using arithmetic operators

```
#include<stdio.h>
int main()
{
	int num;
	printf("\nEnter a number:");
	scanf("%d",&num);
	num=num & 1;
	if(num==0)
		printf("\nNumber is Even");
	else
		printf("\nNumber is Odd");
}
```