1. What is storage class specifier?

Storage class specifiers in C language tells the compiler where to store a variable, how to store the variable, what is the initial value of the variable and life time of the variable.

Syntax: storage_specifier data_type variable _name

Types of storage class specifier in c :
There are four storage class specifier in C Language. They are,
1. Automatic
2. Register
3. Static
4. Extern

S.No.	Storage Specifier	Storage place	Initial / default value	Scope	Life
1	**auto**	CPU Memory	Garbage value	local	Within the function only.
2	**extern**	CPU memory	Zero	Global	Till the end of the main program. Variable definition might be anywhere in the C program
3	**static**	CPU memory	Zero	local	Retains the value of the variable between different function calls.
4	**register**	Register memory	Garbage value	local	Within the function

Note:

- For faster access of a variable, it is better to go for register specifiers rather than auto specifiers.
- Because, register variables are stored in register memory whereas auto variables are stored in main CPU memory.
- Only few variables can be stored in register memory. So, we can use variables as register that are used very often in a C program.

Example Program for Auto variable in C

```
#include
void increment(void);
int main()
{
  increment();
  increment();
  increment();
  increment();
  return 0;
}
void increment(void)
{
  auto int i = 0 ;
  printf ( "%d ", i ) ;
  i++;
}
```

Output: 0 0 0

Example program for static variable in C

```
#include
void increment(void);
int main()
{
  increment();
  increment();
  increment();
  increment();
  return 0;
}
void increment(void)
{
  static int i = 0 ;
  printf ( "%d ", i ) ;
  i++;
}
```

Output: 0 1 2 3

Example program for extern variable in C

The scope of this extern variable is throughout the main program. It is equivalent to global variable. Definition for extern variable might be anywhere in the C program.

```
#inlude
int x = 10 ;
int main( )
{
   extern int y ;
   printf ( "The value of x is %d \n", x ) ;
   printf ( "The value of y is %d",y ) ;
   return 0;
}
int y = 50 ;
```

Output: The value of x is 10
The value of y is 50

Example program for register variable in C:

- Register variables are also local variables, but stored in register memory. Whereas, auto variables are stored in main CPU memory.
- Register variables will be accessed very faster than the normal variables since they are stored in register memory rather than main memory.But, only limited variables can be used as register since register size is very low. (16 bits, 32 bits or 64 bits)

```
#include
int main()
{
   register int i;
   int arr[5];          // declaring array
   arr[0] = 10;         // Initializing array
   arr[1] = 20;
   arr[2] = 30;
   arr[3] = 40;
```

```
  arr[4] = 50;
  for (i=0; i < 5 ; i++)  {
    // Accessing each variable
    printf("value of arr[%d] is %d \n", i, arr[i]);
  }
  return 0;
}
```

Output:

value of arr[0] is 10
value of arr[1] is 20
value of arr[2] is 30
value of arr[3] is 40
value of arr[4] is 50

2. What is static variable?

Ans :There are 3 main uses for the static.

1. **If you declare within a function: It retains the value between function calls.**
2. If it is declared for a function name: By default function is extern..so it will be visible from other files.**if the function declaration is as static..it is invisible for the outer files.**
3. **Static for global variables: By default we can use the global variables from outside files,**

If it is static global..that variable is limited to with in the file.

3. What is difference between static and extern?

Ans:

"The **static** storage class is used to declare an identifier that is a local variable either to a function or a file and that exists and retains its value after control passes from where it was declared. This storage class has a duration that is permanent. A variable declared of this class retains its value from one call of the function to the next. The scope is local. A variable is known only by the function it is declared within or if declared globally in a file, it is known or seen only by the functions within that file. This storage class guarantees that declaration of the variable also initializes the variable to zero or all bits off.

The **extern** storage class is used to declare a global variable that will be known to the functions in a file and capable of being known to all functions in a program. This storage class has a duration that is permanent. Any variable of this class retains its value until changed by another assignment. The scope is global. A variable can be known or seen by all functions within a program. ."

4. What is difference between static local and static global variable?

Ans:

Static global :

Static variable has scope only in the file in which it is declared. it can't be accessed in any other file but its value remains intact if code is running in some other file means lifetime is in complete program .

Static local:

static local variable has scope in that function in which it is declared means it can't be used in other functions in the same file also, means scope is limited to the function in which it is declared while its life time is also through out the program.

5. Can we declare static variable in header file?

Ans:

You can't declare a static variable without defining it as well (this is because the storage class modifiers static and extern are mutuallyexclusive). **A static variable can be defined in a header file,** but this would cause each source file that included the header file to have its own private copy of the variable, which is probably not what was intended.

6. Can we declare main() function as static?

Ans:

No. The C spec actually says somewhere in it that the main function cannot be static.
The reason for this is that static means "don't let anything outside this source file use this object".

These two reasons are why static is a nice thing to have.
Since you can't access static functions from outside the file, how would the OS be able to access the main function to start your program? That's why main can't be static.
Some compilers treat "main" specially and might silently ignore you when you declare it static.

The benefit is that it protects against name collisions in C when you go to link (it would be bad bad bad if you had two globals both named "is_initialized" in different files... they'd get silently merged, unless you made them static). It also allows the compiler to perform certain optimizations that it wouldn't be able to otherwise.

7. Draw memory layout of C program?

Ans :

Refer This link: http://cinterviewquestionandanswer.blogspot.in/2014/01/memory-layout-of-c-programs.html

8.What is volatile variable means?

volatile has nothing to deal with storage class.

volatile just tells the compiler or force the compiler to "not to do the optimization" for that variable. so compiler would not optimize the code for that variable and reading the value from the specified location, not through interal register which holds the previous value.

So**, by declaring variable as volatile.. it gives garrantee that you will get the latest value, which may be alterred by an external event.**
your code may be work fine if haven't declare that variable as volatile, but there may be chance of not getting correct value sometimes.. so to avoid that we should declare variable as volatile.
volatile is generally used when dealing with external events, like interrupts of hardware related pins.

volatile specifies a variable whose value may be changed by processes outside the current program.

volatile qualifierg

• V**ariable may be altered outside program**

• Variable not under control of program

• Variable cannot be optimized

Example. reading adc values.

const voltile means you can not modify or alter the value of that variable in code. only external event can change the value.
controller pins are generally defines as volatile. may be by declaring variable as volatile controller will do "read by pin" not "read by latch"... this is my assumtion. may be wrong...
but still there is lots of confusion when to choose variable as volatile..
A variable should be declared volatile whenever its value could change unexpectedly. **In practice, only three types of variables could change:**
Memory-mapped peripheral registers
Global variables modified by an interrupt service routine
Global variables within a multi-threaded application

9. What does keyword const means?

Ans:
The `const` qualifier explicitly declares a data object as something that cannot be changed. Its value is set at initialization. You cannot use `const` data objects in expressions requiring a modifiable lvalue. For example, a `const` data object cannot appear on the lefthand side of an assignment statement
int const volatile var

10. What do the following declaration means?

const int a; //constant integer val
int const a; //constant value to integer same as 1st one
const int *a; //pointer to constant integer we cant change value but we can assign another addres to pointer a like *a= &b ,but cont change if *a=10 cnt chnge to another val 19
int * const a; //constant pointer to integer we cnt change the pointer but we can chnge the val ...

int const * a const; //we cant change pointer and value alsp

11. Can we use const keyword with volatile variable?

Ans:
Yes. The const modifier means that this code cannot change the value
of the variable, but that does not mean that the value cannot be
changed by means outside this code. For instance, in the example the timer structure was accessed through a volatile const pointer. The function itself did not change the value of the timer, so it was declared const. However, the value was changed by hardware on the computer, so it was declared volatile. If a variable is both const and volatile, the two modifiers can appear in either order.

`volatile` will tell the compiler not to optimise code related the variable, usually when we know it can be changed from "outside", e.g. by another thread.

- `const` will tell the compiler that it is not allowed for the program to modify the variable's value.
- `constvolatile` is a very special thing you'll probably see used exactly 0 times in your life (tm). As is to be expected, it means that the program cannot modify the variable's value, but the value can be modified from the outside, thus no optimisations will be performed on the variable.

Pointer String and array question :

1. What are pointers?

Ans:
A **pointer** is a variable whose value is the address of another variable, i.e., direct address of the memory location. Like any variable or constant, you must declare a pointer before you can use it to store any variable address. The general form of a pointer variable declaration is:

```
type *var-name;
```

Here, **type** is the pointer's base type; it must be a valid C data type and **var-name** is the name of the pointer variable. The asterisk * you used to declare a pointer is the same asterisk that you use for multiplication. However, in this statement the asterisk is being used to designate a variable as a pointer. Following are the valid pointer declaration:

```
int    *ip;    /* pointer to an integer */
double *dp;    /* pointer to a double */
float  *fp;    /* pointer to a float */
char   *ch     /* pointer to a character */
```

The actual data type of the value of all pointers, whether integer, float, character, or otherwise, is the same, a long hexadecimal number that represents a memory address. The only difference between pointers of different data types is the data type of the variable or constant that the pointer points to.

2. What is dangling pointer?

Ans:

A **dangling pointer** points to memory that has already been freed. The storage is no longer allocated. Trying to access it might cause a Segmentation fault.

Common way to end up with a dangling pointer:

```
char* func()
{
   char str[10];
   strcpy(str,"Hello!");
   return(str);
}
//returned pointer points to str which has gone out of scope.

You are returning an address which was a local variable, which would have gone
out of scope by the time control was returned to the calling function.
(Undefined behaviour)
Another common dangling pointer example is an access of a memory location via
pointer, after free has been explicitly called on that memory.

int *c = malloc(sizeof(int));
free(c);
*c = 3; //writing to freed location!
```

3. What is NULL pointer?

Ans:

Pointer assigned value NULL.

Example of null pointer:

1. int *ptr=NULL;

2. int *ptr = new int;

delete ptr;

ptr = NULL; // ptr is null pointer

NULL Pointer is a pointer which is pointing to nothing.

1. NULL pointer points the base address of segment.
2. In case, if you don't have address to be assigned to pointer then you can simply use NULL
3. Pointer which is initialized with NULL value is considered as NULL pointer.
4. NULL is macro constant defined in following header files –
    - stdio.h
    - alloc.h
    - mem.h
    - stddef.h
    - stdlib.h

Null pointer is special reserved value of a pointer. A pointer of any type has such a reserved value. Formally, each specific pointer type (int*, char*) has its own dedicated null-pointer value. Conceptually, when a pointer has that null value it is not pointing anywhere.

4. What is void or generic Pointer?

Ans:

Void pointer or **generic** pointer is a special type of pointer that can be pointed at objects of any data type. A void pointer is declared like a normal pointer, using the void keyword as the pointer's type.

Pointers defined using specific data type cannot hold the address of the some other type of variable i.e., it is incorrect in C++ to assign the address of an integer variable to a pointer of type float.

Example:

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

The above problem can be solved by general purpose pointer called void pointer.

Void pointer can be declared as follows:

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

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

Example:

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

5. What is memory leakage? How can we avoid it?

Ans :

Memory leak occurs when programmers create a memory in heap and forget to delete it.
Memory leaks are particularly serious issues for programs like daemons and servers which by definition never terminate.

```
/* Function with memory leak */
#include

void f()
{
   int *ptr = (int *) malloc(sizeof(int));

   /* Do some work */

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

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

```
/* Function without memory leak */
#include ;

void f()
{
   int *ptr = (int *) malloc(sizeof(int));

   /* Do some work */

   free(ptr);
   return;
}
```

6. What is the size of pointer in 32 bit machine?

Ans:

Sizeof of pointer in 32 bit machine is always 4 bytes.

7. Write a program to find weather machine is 32 bit or 64 bit?

Ans:

```
int main()
{
   int *p = NULL;
   if(sizeof(p) == 4)
      printf("Machine is 32 bit\n");
   else
      printf("Machine is 64 bit\n");
   return 0;
}
```

8. What is array?

Ans:

In C programming, one of the frequently arising problem is to handle similar types of data. For example: If the user want to store marks of 100 students. This can be done by creating 100 variable individually but, this process is rather tedious and impracticable. These type of problem can be handled in C programming using arrays.

An array is a sequence of data item of homogeneous value(same type).

9. What is difference between array and pointer?

Ans:

Pointer	Array
1. A pointer is a place in memory that keeps address of another place inside	1. An array is a single, pre allocated chunk of contiguous elements (all of the same type), fixed in size and location.
2. Pointer can't be initialized at definition.	2. Array can be initialized at definition. Example int num[] = { 2, 4, 5}
3. Pointer is dynamic in nature. The memory allocation can be resized or freed later.	3. They are static in nature. Once memory is allocated , it cannot be resized or freed dynamically.
4. The assembly code of Pointer is different than Array	4. The assembly code of Array is different than Pointer.

An array is an array and a pointer is a pointer, but in most cases array names are *converted* to pointers.

Here is an array:

int a[7]

a contains space for seven integers, and you can put a value in one of them with an assignment, like this:

a[3] = 9;

Here is a pointer:

int *p;

p doesn't contain any spaces for integers, but it can point to a space for an integer. We can for example set it to point to one of the places in the array **a**, such as the first one:

p = &a[0];

What can be confusing is that you can also write this:

p = a;

This does *not* copy the contents of the array **a** into the pointer **p** (whatever that would mean).

Instead, the array name **a** is converted to a pointer to its first element. So that assignment does the same as the previous one.
Now you can use p in a similar way to an array:
p[3] = 17;
The reason that this works is that the array dereferencing operator in C, "[ ]", is defined in terms of pointers. **x[y]** means: start with the pointer **x**, step **y** elements forward after what the pointer points to, and then take whatever is there. Using pointer arithmetic syntax, **x[y]** can also be written as ***(x+y)**.
For this to work with a normal array, such as our **a**, the name **a** in **a[3]** must first be converted to a pointer (to the first element in **a**). Then we step 3 elements forward, and take whatever is there. In other words: take the element at position 3 in the array. (Which is the fourth element in the array, since the first one is numbered 0.)
So, in summary, array names in a C program are (in most cases) converted to pointers. One exception is when we use the **sizeof** operator on an array. If **a** was converted to a pointer in this contest, **sizeof(a)** would give the size of a pointer and not of the actual array, which would be rather useless, so in that case **a** means the array itself.

10. Can we increment the base address of array? Justify your answer.

Ans:
No**,** Because once we initialize the array variable, the pointer points base address only & it's fixed and constant pointer.

11. What is the output of program.

```
int a[5] = {1,2,3,4,5};

int *ptr = (int*) (&a +1);

int *ptr = (int*) (a+1);
```

Ans:

you see, for your program above, a and &a will have the same **numerical value**,and I believe that's where your whole confusion lies.You may wonder that if they are the same,the following should give the **next address** after a in both cases,going by pointer arithmetic:

```
(&a+1) and (a+1)
```

But it's not so!!**Base address** of an array (`a` here) and **Address** of an array are not same! `a` and `&a` might be same numerically ,but they are not the same **type**. `a` is of type `int*` while `&a` is of type `int (*)[5]`,ie , `&a` is a pointer to (address of) and array of size 5.But `a` as you know is the **address of the first element of the array**.Numerically they are the same as you can see from the illustration using ^ below.
But **when you increment these two pointers/addresses, ie as `(a+1)` and `(&a+1)`, the arithmetic is totally different.While in the first case it "jumps" to the address of the next element in the array, in the latter case it jumps by 5 elements as that's what the size of an array of 5 elements.**

12. What is output of the program?

```
int main()
{
        int arr[10];
         int *ptr = arr;
        ptr++;
        arr++;
         return 0;
}
```

Ans:

The statement arr++ will give you lvalue error. Because hear you are trying to increment base address of array and by default base address of array is constant pointer(constant value) so,

arr = arr+1;

i.e according to rule on LHS of assignment operator there always should be lvalue i.e variable not constant.

13. What is string?

Ans:

The string in C programming language is actually a one-dimensional array of characters which is terminated by a null character '\0'. Thus a null-terminated string contains the characters that comprise the string followed by a null.

14. What is difference between these two,

char ptr[] = "string";
char *ptr = "string";

Ans:

The two declarations are not the same.
First one is the array of character i.e. string and second one is the string literals.
-->`char ptr[] = "string";` declares a char array of size `7` and initializes it with the characters `s`,`t`,`r`,`i`,`n`,`g` and `\0`. You are **allowed** to modify the contents of this array.
-->`char *ptr = "string";` declares `ptr` as a char pointer and initializes it with address of *string literal* `"string"` which is ***read-only***. Modifying a string literal is an **undefined behavior**. What you saw(seg fault) is one manifestation of the undefined behavior.

15. Write a program to find size of variable without using sizeof operator?

Ans:

```
#define sizeof(var)     ( (char*)(&var+1) - (char*) (&var))
int main()
{
   int val;
   printf("size of = %d\n", SIZEOF(val));
```

```
   return 0;
}
```

16. Write a program to find sizeof structure without using size of operator?

Ans:

```
#define SIZEOF(var)     ( (char*)(&var+1) - (char*) (&var))

int main()
{
       struct s {
              int a;
              char b;
              int c;
       };
       struct s obj[1];

   printf("size of = %ld\n", SIZEOF(obj));
   return 0;
}
```

17. What is the output of following program?

```
int main()
{
     char str[] = "vishnu";
     printf("%d %d\n",sizeof(str),strlen(str));
}
```

Ans:

7 6

Here char str[] = " 'v'. 'i' ,'s','h','n',u','\0' ";

so sizeof operator always count null character whereas strlen skip null character.

18. Write a program to implement strlen(), strcpy(),strncpy(), strrev(),strcmp() function"?

Ans:

1. strlen:

```
int my_strlen(const char * str)
{
    int count;
    while(* str != '\0') {
        count++;
        str++;
    }
    return 0;
}
```

2. strcpy:

```
void my_strcpy(char * dest, const char* src)
{
    while(* src != '\0') {
        *dest = *src;
        dest++;
        src++;
    }
    *dest = '\0';
}
```

3. strrev:

```
void my_strrev(char *str,size)
{
    int i;
    char temp;
    for(i=0;i<=size/2;i++) {
        temp = str[i];
        str[i] = str[size-i];
        str[size-i] = temp;
    }
}
int main()
{
    char str[10] = "vishnu";
    int len;
    len = strlen(str);
    my_strrev(str,len-1);
    printf("strrev = %s\n",str);
    return 0;
}
```

4. strcmp :

```
void my_strcmp(char * dest, const char* src)
{
      while(*str != '\0' && *dest != '\0') {
       str++;
       dest++;
   }
   return (*src - *dest);
}

int main()
{
   char str[10];
   char dest[10];
   int i;
   i = my_strcmp(dest,src);
   if(i == 0 )
      printf("strings are equal\n");
   if(i < 0)
      printf(" string1 is less than string2\n");
   if(i > 0)
      printf("string2 is greter than string1\n");
   return 0;
}
```

5. strncpy

```
void my_strncpy(char * dest, const char* src,int n)
{
   while(*src != '\0' && n != 0) {
      *dest = *src;
      dest++;
      src++;
      n--;
   }
   while(n) {
      *dest = '\0';
      n--;
   }
}
```

19 . Write a program to implement above function using recursion?

20 . Write a program to check weather string is palindrome or not?

```
#include<stdio.h>
#include<string.h>
int main(void)
{
        char str[50];
        int i,j;
        printf("Enter a string : ");
        gets(str);
        for(i=0,j=strlen(str)-1; i<=j; i++,j--)
                if(str[i]!=str[j])
                        break;
        if(i>j)
                printf("String is a palindrome\n" );
        else
                printf("String is not a palindrome\n" );
        return 0;
}
```

21 . Write a program to count total number of vowel,consonant present in given string?

Ans:

```
int main()
{

   char sentence[80];
   int i, vowels = 0, consonants = 0, special = 0;
   printf("Enter a sentence \n");
   gets(sentence);
       for (i = 0; sentence[i] != '\0'; i++) {

      if ((sentence[i] == 'a' || sentence[i] == 'e' || sentence[i] == 'i' || sentence[i] == 'o' || sentence[i] ==
'u') || (sentence[i] == 'A' || sentence[i] == 'E' || sentence[i] == 'I' || sentence[i] == 'O' || sentence[i] ==
```

```
'U')) {
        vowels = vowels + 1;
        }
        else {
            consonants = consonants + 1;
        }
        if (sentence[i] =='t' ||sentence[i] =='\0' || sentence[i] ==' ') {
            special = special + 1;
        }

    }
    consonants = consonants - special;
      printf("No. of vowels in %s = %d\n", sentence, vowels);
    printf("No. of consonants in %s = %d\n", sentence, consonants);
    return 0;

}
```

22. Write a function to find whether machine is little endian or big endian.?

Ans:

```
void is_little_or_big(int n)
    {
        int num = 0x01;
        char * ptr = (char*)num;
        if(*ptr == 1)
            printf("little endian\n");
        else
            printf("big endian\n");
    }
    or
        void is_little_or_big()
    {
        enum union {
            int a;
            char c[4];
        };
        enum endian obj;
        obj.i = 1;
        if(obj.c[0] == 1)
            printf("Machine is little endian\n");
        else
            printf("machine is big endian\n");
    }
```

Write a program to find occurrence of particular key in given string?

Write a program to move all 0's to one side and 1's on ther side of array?

Write a program to find largest element in an array?

Write a program to find second largest element from array?

Bit Manipulation

1. Write a program to count total number of setbit in 32 bit number?

```
int countset(int num)
{
      int count = 0;
            while (num) {
          if( ((num) & 1) == 1)
              count++;
               num = num >> 1;
                 }
             return count;
}
```

or

```
unsigned int countsetbit(int num)
{
     int count = 0;
    while(num != 0){
count++;
    num = num & (num-1);
   }
     return count;
}
```

2. Write a program to set n th bit in 32 bit number?

```
int setbit(int num, int pos)
{
    num = num | 1 << pos;
}
```

3. Write a program to count total number of reset bit in 32 bit integer?

```
int resetbit(int num)
{
        int count = 0;
        while (num) {
                if( ((num) & 1) == 0)
                        count++;
                num = num >> 1;
          }
          return count;
}
```

4. Write a program to reset nth bit in 32 bit number?

```
int resetbit(int num, int pos)
{
        num = num &  ~(1 << pos);
}
```

5. Write a program to swap nibble of a 1byte data?

6. Write a program to swap two variable using bitwise operator?

```
void swap(int a, int b)
{
       a = a ^ b;
       b = a ^  b;
       a = a ^ b;
}
```

7. Write a program to find number is even or odd?

```
void evnodd(int num)
{
     if( (num) & (1) )       //if(1) true odd num else even num
        printf("odd");
    else
        printf("even");
}
```

8. Write a program to find number is power of 2 or not?

```
void  power(int num)
{
     if( !( (num) & (num-1) ) )
         printf("power of 2");
     else
         printf("num is not power of 2");
}
```

9 Write a function to swap even bits with consecutive odd bits in a number.

e.g. bo swapped with b1,b2 sawpped with b3 and so on.

Given an unsigned integer, swap all odd bits with even bits. For example, if the given number is 23 (**0**0**1**0**1**1**1**), it should be converted to 43 (0**0**1**0**1**0**1**1**). Every even position bit is swapped with adjacent bit on right side (even position bits are highlighted in binary representation of 23), and every odd position bit is swapped with adjacent on left side.
If we take a closer look at the example, we can observe that we basically need to right shift (>>) all even bits (In the above example, even bits of 23 are highlighted) by 1 so that they become odd bits (highlighted in 43), and left shift (<<) all odd bits by 1 so that they become even bits. The following solution is based on this observation. The solution assumes that input number is stored using 32 bits.
Let the input number be x
1) Get all even bits of x by doing bitwise and of x with 0xAAAAAAAA. The number 0xAAAAAAAA is a 32 bit number with all even bits set as 1 and all odd bits as 0.
2) Get all odd bits of x by doing bitwise and of x with 0x55555555. The number 0x55555555 is a 32 bit number with all odd bits set as 1 and all even bits as 0.
3) Right shift all even bits.
4) Left shift all odd bits.
5) Combine new even and odd bits and return.

```
// C program to swap even and odd bits of a given number
#include
unsigned int swapBits(unsigned int x)
{
    // Get all even bits of x
    unsigned int even_bits = x & 0xAAAAAAAA;
    // Get all odd bits of x
    unsigned int odd_bits  = x & 0×55555555;
    even_bits >>= 1;  // Right shift even bits
    odd_bits <<= 1;   // Left shift odd bits
    return (even_bits | odd_bits); // Combine even and odd bits
}
// Driver program to test above function
int main()
```

```
{
    unsigned int x = 23; // 00010111
    // Output is 43 (00101011)
    printf("%u ", swapBits(x));
    return 0;
}
```

Output:

```
43
```

10. Write a function to set a particular bit.

```
unsigned int setbit(unsigned inr num,int pos)
{
 num = num | (1 << pos);
 return num;
}
```

11. Write a function to clear a particular bit.

```
unsigned int clear(unsigned inr num,int pos)
{
 num = num & ~ (1 << pos);
 return num;
}
```

12. Write a function to toggle particular bit.

```
unsigned int togglebit(unsigned inr num,int pos)
{
 num = num ^ (1 << pos);
 return num;
}
```

13. Write a function to swap even bits with consecutive odd bits in a number.

e.g. b0 swapped with b1, b2 swapped with b3 and so on.

```
unsigned int swap_bits(unsigned int num)
{
  return (num >> 1 & 0x55555555) | (num << 1 & 0xAAAAAAAA);
}
```

14. Write a function to swap odd bits in a number.

e.g. b1 swapped with b3, b5 swapped with b7 and so on.

```
unsigned int swap_odd_bits(unsigned int num)
{

 return (num >> 2 & 0x22222222) |
```

```
        (num << 2 & 0x88888888) |
        ( num   & 0x55555555) ;
}
```

15. Write a function to swap even bits in a number.

e.g. b0 swapped with b2, b4 swapped with b6 and so on.

```
unsigned int swap_even_bits(unsigned int num)
{
return (num >> 2 & 0x11111111) |
   (num << 2 & 0x44444444) |
   ( num   & 0xAAAAAAAA);}
```

16. Write a function to find out the number of 1s in a number.

```
unsigned int num_of_ones(unsigned int num)
{
 if( (count_ones(num) & 1)
  return ODD;
 else
  return EVEN;
}
```

17. Write a function for finding the first lowest bit set in a number.

```
unsigned int first_lowest_bit(unsigned num)
{
 int count =0;
 while(num) {
  count ++;
  if( (num) & 1 == 1)
   break;
  num = num >> 1;
 }
 return count;
}
```

18. Write a function for finding the higest bit set in a number.

```
unsigned int first_highest_bit(unsigned num)
{
 int count =0;
 while(num) {
  count ++;
  if( (num & (1 << 31) ) == 1)
   break;
  num = num << 31;
 return count;
}
```

19 Write a function for reversing the bits in a number.

```
unsigned int reverse_bit(unsigned num)
{
 unsigned int NO_OF_BITS = sizeof(num) * 8;
 unsigned int temp,rev=0;
```

```
 for(i=0; i <= NO_OF_BITS -1 ;i++) {
  if(temp) {
   rev |= (1 << ((NO_OF_BITS-1)-i);
 }
 return rev;
}
```

20. Write a code to extract nth to mth bit, where n<m

```
(num >> n) & ~(~ 0 << (m-n+1))
```

21. write a code for toggling nth to m bits,where n < m

```
num ^ ((~ 0 << n) & ( ~0 >> (31-m)))
```

or

```
num ^ (( 1 << n) & ( 1 >> (31-m)))
```

22. Write a code for setting nth to mth bit, where n < m

```
num | ((~0 << n) & (~0 >>(31-m)))
```

23. write a code for clearing nth to mth bit, where n < m

```
num & ~((~0 << n) & (~0 >> (31-m)))
```

Link List Question:

1. How to check whether linked list is circular or not.

Ans:

```
struct node {

 int data;
 struct node *next;
};
struct node *head = NULL;

void checkcircular(struct node *head)
{
 struct node * slow = head;
 struct node * fast = head;
 while( fast && fast->next)

{
 if(slow == fast->next->next)
```

```
{
   printf("Circular\n');
     break;
   }
    else {
     slow = slow->next;
     fast = fats->next->next;
   }
  }
}
```

2. How would you find a loop in a singly linked list?

```
struct node {
    int data;
    struct node *next;
};

struct node *head = NULL:

void detectloop(struct node * head)
{
 struct node * slow = head;
 struct node * fast = head;

 while(slow && fast && fast->next) {
    slow = slow->next;
  fast = fast->next->next;
  if (slow == fast) {
  printf("Loop detected\n');
  break;
}
 }
}
```

3. Write a C function to print the middle of a given linked list.

```
struct node {
    int data;
    struct node *next;
};

void findmiddle(struct node *head)
```

```
{
   struct node * slow = head;
   struct head * fast = head;
   while (fast != NULL && fast->next != NULL) {
      slow = slow->next;
      fast = slow->next->next;
   }
   printf(" Middle element is %d\n", slow->data);
}
```

4. Write a c program to get the intersection point of two singly linked lists.

```
struct node {
   int data;
   struct node *next;
};

struct node * head = NULL:

int count_node(struct node * head)
{
   int count = 0;
   struct node * current = head;
   while (current != NULL) {
      count ++;
      current = current->next;
   }
   return count;
}

struct get_intersection_mod(int d,struct node * head1, struct node * head2)
{
   struct node current1 = head1;
   struct node current2 = head2;

   for(i=0;i
   }
   while(cuttent1 != NULL && current2 != NULL) {
      if(current1 == current2) {
         printf(" intersection node =%d\n", current1->data);
```

```
            break;
         }
         current1 = current1->next;
         current2 = current2->next;
    }
}
void get_intersecton(struct node * head1, struct node * head2)
{
    struct node *current = head1;
    struct node *current = head2;
    int c1, c2;
    c1 = count_node(current1);
    c2 = count_node(current2);
    if(c1 > c2) {
        d= c1 -c2;
        get_intersection_mod(d,head1,head2);
    }
    else {
        d = c2 -c1;
        get_intersection_mod(d, head1,head2);
    }
}
```

,,,

4)how to access array elements without using index and

incrementing pointer variable?

```
#include <stdio.h>

int main ()
{
   /* an array with 5 elements */
   double balance[5] = {1000.0, 2.0, 3.4, 17.0, 50.0};
   double *p;
   int i;

   p = balance;

   /* output each array element's value */

   printf( "Array values using pointer\n");
   for ( i = 0; i < 5; i++ )
   {
       printf("*(p + %d) : %f\n",  i, *(p + i) );
   }

   printf( "Array values using balance as address\n");
   for ( i = 0; i < 5; i++ )
   {
       printf("*(balance + %d) : %f\n",  i, *(balance + i) );
   }
```

```
   return 0;
}
```

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

```
Array values using pointer
*(p + 0) : 1000.000000
*(p + 1) : 2.000000
*(p + 2) : 3.400000
*(p + 3) : 17.000000
*(p + 4) : 50.000000
Array values using balance as address
*(balance + 0) : 1000.000000
*(balance + 1) : 2.000000
*(balance + 2) : 3.400000
*(balance + 3) : 17.000000
*(balance + 4) : 50.000000
```

the differences between strcpy() and memcpy():?

- memcpy() copies specific number of bytes from source to destinatio in RAM, where as strcpy() copies a constant / string into another string.

- memcpy() works on fixed length of arbitrary data, where as strcpy() works on null-terminated strings and it has no length limitations.

- memcpy() is used to copy the exact amount of data, whereas strcpy() is used of copy variable-length null terminated strings.

Structure Padding

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

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

```
struct student
{
    int id1;
    int id2;
    char a;
    char b;
    float percentage;
```

};

- As per C concepts, int and float datatypes occupy 4 bytes each and char datatype occupies 1 byte for 32 bit processor. So, only 14 bytes (4+4+1+1+4) should be allocated for above structure.
- But, this is wrong. Do you know why?
- Architecture of a computer processor is such a way that it can read 1 word from memory at a time.
- 1 word is equal to 4 bytes for 32 bit processor and 8 bytes for 64 bit processor. So, 32 bit processor always reads 4 bytes at a time and 64 bit processor always reads 8 bytes at a time.
- This concept is very useful to increase the processor speed.
- To make use of this advantage, memory is arranged as a group of 4 bytes in 32 bit processor and 8 bytes in 64 bit processor.
- Below C program is compiled and executed in 32 bit compiler. Please check memory allocated for structure1 and structure2 in below program.

Example program for structure padding in C:

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

/*  Below structure1 and structure2 are same.
    They differ only in member's allignment */

struct structure1
{
       int id1;
       int id2;
       char name;
       char c;
       float percentage;
};

struct structure2
{
       int id1;
       char name;
       int id2;
       char c;
       float percentage;
};

int main()
{
    struct structure1 a;
    struct structure2 b;

    printf("size of structure1 in bytes : %d\n",
            sizeof(a));
    printf ( "\n   Address of id1        = %u", &a.id1 );
    printf ( "\n   Address of id2        = %u", &a.id2 );
    printf ( "\n   Address of name       = %u", &a.name );
    printf ( "\n   Address of c          = %u", &a.c );
    printf ( "\n   Address of percentage = %u",
                   &a.percentage );
```

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

Output:

size of structure1 in bytes : 16

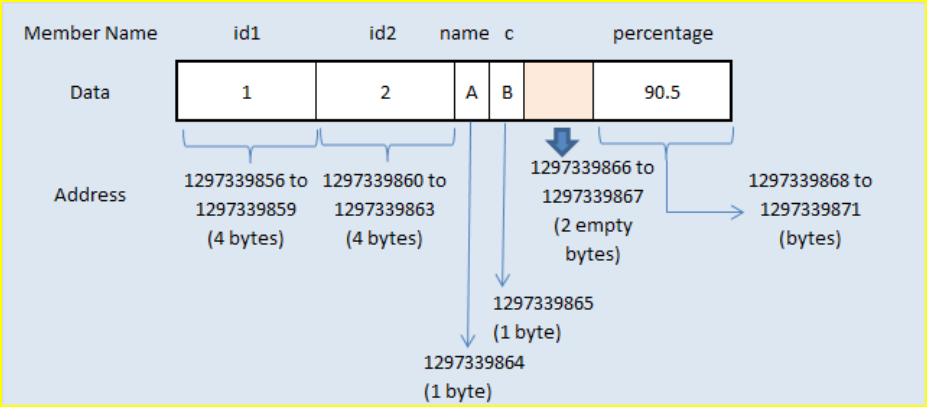
Address of id1 = 1297339856
Address of id2 = 1297339860
Address of name = 1297339864
Address of c = 1297339865
Address of percentage = 1297339868

size of structure2 in bytes : 20

Address of id1 = 1297339824
Address of name = 1297339828
Address of id2 = 1297339832
Address of c = 1297339836
Address of percentage = 1297339840

Structure padding analysis for above C program:

Memory allocation for structure1:

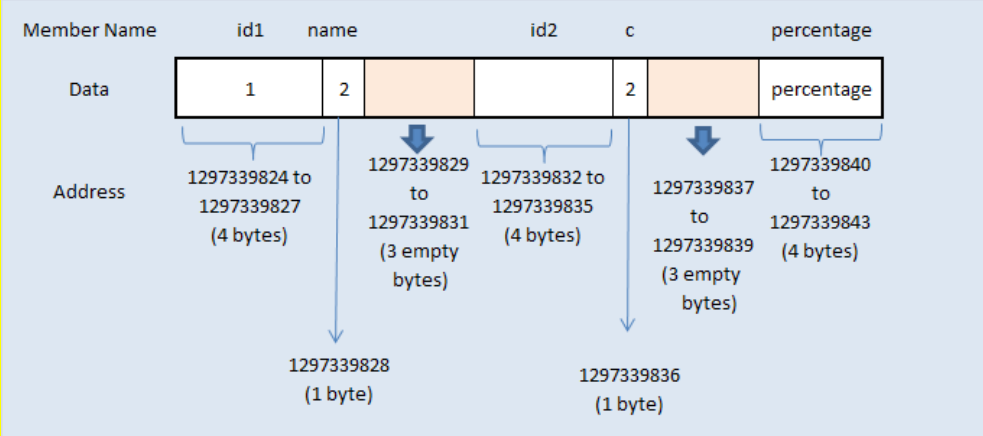


- In above program, memory for structure1 is allocated sequentially for first 4 members.
- Whereas, memory for 5th member “percentage” is not allocated immediate next to the end of member “c”.
- There are only 2 bytes remaining in the package of 4 bytes after memory allocated to member “c”.
- Range of this 4 byte package is from 1297339864 to 1297339867.
- Addresses 1297339864 and 1297339865 are used for members “name and c”. Addresses

1297339866 and 1297339867 only is available in this package.

- But, member “percentage” is datatype of float and requires 4 bytes. It can’t be stored in the same memory package as it requires 4 bytes. Only 2 bytes are free in that package.
- So, next 4 byte of memory package is chosen to store percentage data which is from 1297339868 to 1297339871.
- Because of this, memory 1297339866 and 1297339867 are not used by the program and those 2 bytes are left empty.
- So, size of structure1 is 16 bytes which is 2 bytes extra than what we think. Because, 2 bytes are left empty.

Memory allocation for structure2:



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

How to avoid structure padding in C?

- #pragma pack (1) directive can be used for arranging memory for structure members very

next to the end of other structure members.

- VC++ supports this feature. But, some compilers such as Turbo C/C++ does not support this feature.
- Please check the below program where there will be no addresses (bytes) left empty because of structure padding.

Example program to avoid structure padding in C:

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

/*  Below structure1 and structure2 are same.
    They differ only in member's allignment */

#pragma pack(1)
struct structure1
{
       int id1;
       int id2;
       char name;
       char c;
       float percentage;
};

struct structure2
{
       int id1;
       char name;
       int id2;
       char c;
       float percentage;
};

int main()
{
    struct structure1 a;
    struct structure2 b;

    printf("size of structure1 in bytes : %d\n",
                    sizeof(a));
    printf ( "\n   Address of id1        = %u", &a.id1 );
    printf ( "\n   Address of id2        = %u", &a.id2 );
    printf ( "\n   Address of name       = %u", &a.name );
    printf ( "\n   Address of c          = %u", &a.c );
    printf ( "\n   Address of percentage = %u",
                    &a.percentage );

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

Output:

```
size of structure1 in bytes : 14

Address of id1 = 3438103088
Address of id2 = 3438103092
Address of name = 3438103096
Address of c = 3438103097
Address of percentage = 3438103098

size of structure2 in bytes : 14

Address of id1 = 3438103072
Address of name = 3438103076
Address of id2 = 3438103077
Address of c = 3438103081
Address of percentage = 3438103082
```

Padding aligns structure members to "natural" address boundaries - say, `int` members would have offsets, which are `mod(4) == 0` on 32-bit platform. Padding is on by default. It inserts the following "gaps" into your first structure:

```
struct mystruct_A {
    char a;
    char gap_0[3]; /* inserted by compiler: for alignment of b */
    int b;
    char c;
    char gap_1[3]; /* -"-: for alignment of the whole struct in an array */
} x;
```

structure Packing, on the other hand prevents compiler from doing padding - this has to be explicitly requested - under GCC it's `__attribute__((__packed__))`, so the following:

```
struct __attribute__((__packed__)) mystruct_A {
    char a;
    int b;
    char c;
};
```

would produce structure of size `6` on a 32-bit architecture.

A note though - unaligned memory access is slower on architectures that allow it (like x86 and amd64), and is explicitly prohibited on *strict alignment architectures* like SPARC.

Structure packing suppresses structure padding, padding used when alignment matters most, packing used when space matters most.

Some compilers provide `#pragma` to suppress padding or to make it packed to n number of bytes. Some provide keywords to do this. Generally pragma which is used for modifying structure padding will be in the below format (depends on compiler):

```
#pragma pack(n)
```

For example ARM provides the `__packed` keyword to suppress structure padding. Go through your compiler manual to learn more about this.

So a packed structure is a structure without padding.

Generally packed structures will be used

- to save space
- to format a data structure to transmit over network using some protocol (this is not a good practice of course because you need to deal with endianness)

If you'd like a more detailed explanation on structure padding and packing, please refer to my blog (original link is now a link spammer. Try the Wayback Machine version instead.).

22. What is extern and static function in C?

- By default, any function that is defined in a C file is extern. These functions can be used in any other source file of the same project which has many other files.
- When we declare/define a function as static, these functions can't be used in other files of the same project.
- Also, if we want to use the same function name in different files of the same project, we can use static function which won't through any compilation error for duplicate function name.

123. What is the difference between pointer and array in C?

- Array is a collection of variables belongings to the same data type. We can store group of data of same data type in an array.
- Pointer is a single variable that stores the address of other object/variable.

124. Can a variable be both volatile and constant in C?

Yes. A variable can be declared as both volatile and constant in C.

Const modifier does not allow changing the value of the variable by internal program. But, it does not mean that value of const variable should not be changed by external code. So, a variable can be both volatile and constant in C.

125. What is the use of main() function in C?

main() function is the function from where execution of any C program begins. So, main() function is mandatory for any C program.

119. What is the scope of local, global and environment variables in C?

- The scope of **local variables** will be within the function only. These variables are declared within the function and can't be accessed outside the function.

- The scope of **global variables** will be throughout the program. These variables can be accessed from anywhere in the program. This variable is defined outside the main function. So, this variable is visible to main function and all other sub functions.
- **Environment variable** is a variable that will be available for all C applications and C programs. We can access these variables from anywhere in a C program without declaring and initializing in an application or C program.

120. Why are we using pointers in C?

- C Pointer is a variable that stores/points the address of another variable.
- C Pointer is used to allocate memory dynamically (i.e. at run time)
- simulating call by reference
- returning more than one value from function
- we can access dynamically allocated memory
- implementing data structures like linked list trees graphs
- improve efficiency

121. What is static variable in C?

Static variable is a variable that retains it's value between different function calls.

117. What is inline function in C?

- A normal function becomes inline function when function prototype of the function is prepended with keyword "inline".
- The property of inline function is, compiler inserts the entire body of the function in the place where inline function name is used in the program.
- Advantage of inline function is, it does not require function call and does not return anything from the function.
- Disadvantage of inline function is, it increases file size as same function code is copied again and again in the program wherever it is called.

118. Is it possible to print "Hello World" without semicolon in C? How?

Yes. It is possible to print "Hello World" without semicolon in C program. Please refer below example program.

This program prints "Hello World" in console window and returns 11 which is the value of string length.

```
#include <stdio.h>
int main()
{
if(printf("Hello World"))
{
/* Do nothing */
```

}

}

97. What is wild pointer in C?

Uninitialized pointers are called as wild pointers in C which points to arbitrary (random) memory location. This wild pointer may lead a program to behave wrongly or to crash.

98. What is file pointer in C?

- File pointer is a pointer which is used to handle and keep track on the files being accessed. A new data type called “FILE” is used to declare file pointer. This data type is defined in stdio.h file. File pointer is declared as FILE *fp. Where, ‘fp’ is a file pointer.
- fopen() function is used to open a file that returns a FILE pointer. Once file is opened, file pointer can be used to perform I/O operations on the file. fclose() function is used to close the file.

99. When can void pointer and null pointer be used in C?

Void pointer is a generic pointer that can be used to point another variable of any data type. Null pointer is a pointer which is pointing to nothing.

100. What is const pointer in C?

Const pointer is a pointer that can’t change the address of the variable that is pointing to. Once const pointer is made to point one variable, we can’t change this pointer to point to any other variable. This pointer is called const pointer.

101. Is pointer arithmetic a valid one? Which arithmetic operation is not valid in pointer? Why?

Pointer arithmetic is not valid one. Pointer addition, multiplication and division are not allowed as these are not making any sense in pointer arithmetic.

But, two pointers can be subtracted to know how many elements are available between these two pointers.

102. Is void pointer arithmetic a valid one? Why?

Arithmetic operation on void pointer is not valid one. Void pointer is a generic pointer. It is not referring int, char or any other data type specifically. So, we need to cast void pointer to specific type before applying arithmetic operations.

Note:

- Pointer arithmetic itself is not valid one. Pointer addition, multiplication and division are not allowed as these are not making any sense in pointer arithmetic.
- But, two pointers can be subtracted to know how many elements are available between these two pointers.

103. What is the difference between null and zero?

NULL is a macro which is defined in C header files. The value of NULL macro is 0.
It is defined in C header files as below.
#define NULL (void *) 0;

- NULL is used for pointers only as it is defined as (void *) 0. It should not be used other than pointers. If NULL is assigned to a pointer, then pointer is pointing to nothing.
- 0 (zero) is a value.

104. What is the difference between null pointer and uninitialized pointer in C?

Null pointer is a pointer which is pointing to nothing. Null pointer points to empty location in memory. Value of null pointer is 0. We can make a pointer to point to null as below.

```
int *p =  NULL;
char *p = NULL;
```

Uninitialized pointers are called as wild pointers in C which points to arbitrary (random) memory location. This wild pointer may lead a program to behave wrongly or to crash.

105. Can array size be declared at run time?

Array size can't be declared at run time. Size of array must be known during compilation time. So, array size should be declared before compilation.

- Correct example: char array [10];
- Wrong example: char array[i];
- Always, Array subscript should be positive and it should not be either negative or any variable name.
- If we really don't know the size of an array, we can use dynamic memory allocation concepts in C which uses malloc(), calloc() functions to allocate memory during execution time.

106. What is memory leak in C?

- Memory leak occurs when memory is allocated but not released back to the operating system. Memory leakage increases unwanted memory usage. So, it reduces performance of the computer by reducing available memory.
- In modern operating systems, memory is released back to the operating system when application terminates.

107. What happens when we try to access null pointer in C?

NULL pointer is pointer that is pointing to nothing (No memory location). Accessing null pointer in C may lead a program to crash. So, Null pointer should not be accessed in a program.

108. What is meant by segmentation fault or memory fault in C?

Segmentation fault is a fault that occurs because of illegal/invalid memory access.
Illegal memory access means, When a program tries to access a memory location that is not allowed or when a program tries to access a memory location in a way that is not allowed.

109. What is meant by core dump in C?

- **Core dump or core is a file, generated when a program is crashed or terminated abnormally because of segmentation fault or some other reason. Information of the memory used by a process is dumped in a file called core. This file is used for debugging purpose. Core dump has file name like "core.<process_id>"**
- Core dump file is created in current working directory when a process terminates abnormally. Core dump is a typical error occurs because of illegal memory access.
- Core dump is also called as memory dump, storage dump or dump.

73. Where should type cast function not be used in C?

- Type cast function should not be used in const and volatile declaration.
- Because, constant value can't be modified by the program once they are defined.
- And, volatile variable values might keep on changing without any explicit assignment by the program as operating system will be modifying these values.

74. How many arguments can be passed to a function in C?

- Any number of arguments can be passed to a function. There is no limit on this. Function arguments are stored in stack memory rather than heap memory. Stack memory allocation is depending on the operating system.
- So, any number of arguments can be passed to a function as much as stack has enough memory. Program may crash when stack overflows.

75. What is static function in C?

All functions are global by default in a C program/file. But, static keyword makes a function as a local function which can be accessed only by the program/file where it is declared and defined. Other programs/files can't access these static functions.

79. What is the difference between exit() and return() in C?

- exit() is a system call which terminates current process. exit() is not an instruction of C language.
- Whereas, return() is a C language instruction/statement and it returns from the current function (i.e. provides exit status to calling function and provides control back to the calling function).

80. What is the use of "#define" in C?

#define is a pre-processor directive which is used to define constant value. This constant can be any of the basic data types.

88. Is there any inbuilt library function in C to remove leading and trailing spaces from a string? How will you remove them in C?

- There is no inbuilt function to remove leading and trailing spaces from a string in C. We need to write our own function to remove them.
- We need to check first non-space character in given string. Then, we can copy that string from where non space character is found. Then, we can check whether any spaces are available in copied string from the end of the string. If space is found, we can copy '\0' in that space until any character is found. Because, '\0'indicates the end of the string. Now, we have removed leading and trailing spaces in a given string.

89. What is the difference between strcpy() & strncpy() functions in C?

- strcpy( ) function copies whole content of one string into another string. Whereas, strncpy( ) function copies portion of contents of one string into another string.
- If destination string length is less than source string, entire/specified source string value won't be copied into destination string in both cases.

85. How to check whether macro is defined or not in a C program?

"#ifdef" pre-processor directive checks whether particular macro is defined or not in a C program.

86. What is the difference between memcpy() & strcpy() functions in C?

- memcpy() function is used to copy a specified number of bytes from one memory to another. Whereas, strcpy() function is used to copy the contents of one string into another string.
- memcpy() function acts on memory rather than value. Whereas, strcpy() function acts on value rather than memory.

87. What is the difference between memcpy() & memmove() functions in C?

- memcpy() function is is used to copy a specified number of bytes from one memory to another.
- memmove() function is used to copy a specified number of bytes from one memory to another or to overlap on same memory.
- Difference between memmove() and memcpy() is, overlap can happen on memmove(). Whereas, memory overlap won't happen in memcpy() and it should be done in non-destructive way.

- **110. Can a pointer be freed more than once in C? What happens if do so? Or can a pointer be freed twice in C?**

1st scenario: After freeing a pointer in a C program, freed memory might be reallocated by some other or same program luckily. In this scenario, freeing the same pointer twice won't cause any issue.

2nd scenario: We can free a pointer. Then, we can allocate memory for same pointer variable. Then, we can use it and free it again. This is also not an issue.

3rd scenario: If we free the same pointer second time without reallocating memory to that pointer, then what happens? As per ANSI/ISO C standard, this is undefined behaviour. This undefined behaviour may cause anything to the program that we do not expect to happen.

111. What is the size of int pointer and char pointer in C?

- Pointer variable size is not depending on data type as pointer always stores the address of other variable which is always integer data type.
- So, any pointer (int, char, double, etc) size will be 2 for 16 bit processor, 4 for 32 bit processor and 8 for 64 bit processor.
- sizeof() operator can be used to evaluate size of a variable/pointer in C.

112. How will you print the value and address of a pointer variable (example int *p) in C?

- We can use printf ("%x", p); statement to print the address that pointer "p" stores.
- We can use printf ("%d", *p); statement to print the value of the pointer variable.

113. How will you print the value and address of a normal variable (example int p) in C?

We can use printf ("%x", &p); statement to print the address that pointer "p" stores.

We can use printf ("%d", p); statement to print the value of the normal variable.

114. What are library functions and their use in C language? Can we write our own functions and include them in C library?

- Library functions in C language are inbuilt functions which are grouped together and placed in a common place called library. The use of library function is to get the pre-defined output instead of writing our own code to get those outputs.
- Yes. We can write our own functions and include them in C library.

115. Can variable name be start with underscore in C?

Yes. A variable name can start with underscore in C programming language

94. What is NULL in C?

NULL is a macro which is defined in C header files. The value of NULL macro is 0. It is defined in C header files as below.

#define NULL (void *) 0;

NULL is used for pointers only as it is defined as (void *) 0. It should not be used other than pointers. If NULL is assigned to a pointer, then pointer is pointing to nothing.

95. What is void pointer in C?

- Void pointer is a generic pointer that can be used to point another variable of any data type.
- Void pointer can store the address of variable belonging to any of the data type. So, void

pointer is also called as general purpose pointer.

Note:
int pointer can be used to point a variable of int data type and char pointer can be used to point a variable of char data type.

96. What is dangling pointer in C?

When a pointer is pointing to non-existing memory location is called dangling pointer.

91. What is the difference between array and string in C?

- Array can hold any of the data types. But, string can hold only char data type.
- Array size must be a constant value. So, array size can't be changed once declared. But, string size can be modified using char pointer.
- Array is not ended with null character by default. But, string is ended with null ('\0') character by default.

92. What is pointer in C?

Pointer is a variable that stores/points the address of another variable. Normal variable stores the value of the variable whereas pointer variable stores the address of the variable.

Syntax: data_type *var_name;
Where, * is used to denote that "p" is pointer variable and not a normal variable.

93. What is null pointer in C?

Null pointer is a pointer which is pointing to nothing. Null pointer points to empty location in memory. Value of null pointer is 0. We can make a pointer to point to null as below.

int *p = NULL;
char *p = NULL;

3 Macros

A macro is a fragment of code which has been given a name. Whenever the name is used, it is replaced by the contents of the macro. There are two kinds of macros. They differ mostly in what they look like when they are used. Object-like macros resemble data objects when used, function-like macros resemble function calls

Difference between const & const volatile

The `volatile` qualifiers ensures that these optimizations are not performed by the compiler.

If we declare a variable as `volatile` every time the fresh value is updated
If we declare a variable as `const` then the value of that variable will not be changed

An object marked as `const volatile` will not be permitted to be changed by the code (an error will be raised due to the `const` qualifier) - at least through that particular name/pointer.

The `volatile` part of the qualifier means that the compiler cannot optimize or reorder access to the object.

In an embedded system, this is typically used to access hardware registers that can be read and are updated by the hardware, but make no sense to write to (or might be an error to write to).

3.What is big endian and little endian?

Big-endian and little-endian are terms that describe the order in which a sequence of bytes are stored in computer memory.

Big Endian

In big endian, you store the most significant byte in the smallest address. Here's how it would look:

Address	Value
1000	90
1001	AB
1002	12
1003	CD

Little Endian

In little endian, you store the *least* significant byte in the smallest address. Here's how it would look:

Address	Value
1000	CD
1001	12
1002	AB
1003	90

Notice that this is in the reverse order compared to big endian. To remember which is which, recall whether the least significant byte is stored first (thus, little endian) or the most significant byte is stored first (thus, big endian).

Notice I used "byte" instead of "bit" in least significant bit. I sometimes abbreciated this as LSB and MSB, with the 'B' capitalized to refer to byte and use the lowercase 'b' to represent bit. I only refer to most and least significant byte when it comes to endianness.

Is there a quick way to determine endianness of your machine? There are n no. of ways for determining endianness of your machine. Here is one quick way of doing the same.

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

Diffrence b/w union and structure

Structure	Union
1.**The keyword struct is used to define a structure**	1. **The keyword union is used to define a union.**
2. When a variable is associated with a structure, the compiler allocates the memory for each member. The size of structure is greater than or equal to the sum of sizes of its members. The smaller members may end with unused slack bytes.	**2. When a variable is associated with a union, the compiler allocates the memory by considering the size of the largest memory. So, size of union is equal to the size of largest member.**
3. **Each member within a structure is assigned unique storage area of location.**	3. **Memory allocated is shared by individual members of union.**
4. The address of each member will be in ascending order This indicates that memory for each member will start at different offset values.	4. The address is same for all the members of a union. This indicates that every member begins at the same offset value.
5 Altering the value of a member will not affect other members of the structure.	5. Altering the value of any of the member will alter other member values.
6. **Individual member can be accessed at a time**	6. **Only one member can be accessed at a time.**
7. Several members of a structure can initialize at once.	7. **Only the first member of a union can be initialized.**

Function pointer

A function pointer is a variable that stores the address of a function that can later be

called through that function pointer. This is useful because functions encapsulate behavior. For instance, every time you need a particular behavior such as drawing a line, instead of writing out a bunch of code, all you need to do is call the function. But sometimes you would like to choose different behaviors at different times in essentially the same piece of code. Read on for concrete examples.

Benefits of Function Pointers

- Function pointers provide a way of passing around instructions for how to do something
- You can write flexible functions and libraries that allow the programmer to choose behavior by passing function pointers as arguments
- This flexibility can also be achieved by using classes with virtual functions