1) What are Pointers?

Pointers are special kind of variables that can store the address of another variable.

2) What are uninitialised pointers and NULL Pointer

When a pointer variable is declared, it is allocated a memory location just like any other variable. That memory location may have any garbage value before the pointer is initialised. Such Pointers are called uninitialised pointers. It is invalid to dereference an uninitialised pointer. The compiler will not complain, but runtime will throw an exception.

When a pointer is not pointing to any thing, that is the address value in a pointer variable is equivalent to "0" , then it is called a NULL pointer. Just like un-initialised pointers, NULL pointers are also invalid for dereferencing.

3) Pointer syntax & Pointer Arithmetic (reference, dereference , addition, subtraction) , precedence

Pointers are declared using the below syntax :

<data type> * p1;
<data type> *p2;

We use the reference operator & to access the address of a variable.

<data type> var;

Here var is a variable of type <data type>. To assign the address of var to p1, we use the below syntax :
p1 = &var;

Once a pointer made to point to an address location, we can access the value at that address using dereferencing pointer *.

*p1 = 100;
Here the below two statements will give the same output "100".
printf("%d", *p1);
printf("%d",var);

Not all arithmetic operators are valid when it comes to pointers. We can use Addition/ Subtraction or relational operations on Pointer variables, with some rules.

A Pointer can be incremented or decremented by some integer value. When we do that, the pointer value will not exactly be incremented by the integer value. The resulting value will depend on the data type of the pointer.

(p1+ 1) will always result in an address location, which is as many number of bytes away from p1 as the size of the data type the pointer is pointing to.
Eg ., If the pointer is an integer pointer,p1+ 1 will increment the address location by 4 bytes as the size of integer data type on our processor is 4 bytes. In some compilers it may take 2 bytes ,it is system dependent. The size of a datatype is called **scale factor** for that data type.

Similarly, p1 – 1 will decrement the address location to an address 4 bytes before the address p1 was pointing to.
Addition of two pointers is not valid, it will give some value which may not be a valid address location. However we can add two pointers if we are going to subtract another pointer in the same expression, something like p1 + p2 – p3.
We can decrement one pointer from another, to find the number of items in between. So p1 – p2 is valid.
Pointers can be compared using relational operations, p1 > p2,p1 < p2 , p1 == p2 and p1 != p2 are valid.
Shorthand operators like ++,--,+=,-= are allowed on pointers provided the above rules are followed.

4) Call by Value / Call by Reference (Swapping example)

There are two different ways of passing arguments to a function: Call by Value, Call by Reference.

Call by Value :
- When a vaiable is passed by Value to a function, a seperate copy is created for that variable at a different location , with the same value. Which means, only value of the variable is passed to the function, not the actual variable.
- The called function will use these copies as its parameters for manipulation.
- Whatever modifications made to these parameters will not be reflected to the actual arguments sent by the calling function.
- Unless specified as a pointer, all the primary data types like int, float, char, long etc are passed as “Call by Value” arguments.

Call by Reference:
- When arguments are passed by Reference, address of each argument is passed to the function, instead of just the value.
- The called function will use the addresses, or pointers, to manipulate the arguments directly.
- As the function is manipulating the pointers any changes made to these parameters will reflect to the actual variables sent as arguments.
- If we are sending an array name as an argument, it is passed as “Call by Reference” argument, as we are passing the pointer created with array name and not the array itself.
- If we want to send any primary data type variables by value, we have to use the pointer syntax, inorder to pass its address.

Swapping two integer variable values :
Using Call by Value :

```
int main()
{
        int a,b;
        a = 10;
        b = 20;
        Swap(a,b);
        printf(“a = %d\nb = %d\n”,a,b);
}

Swap(int x,int y)
{
        int z;
```

```
        z = x;
        x = y;
        y = z;
        printf("x = %d\ny = %d\n",x,y);

}
```

this function will give the following output :
x = 20
y = 10
a = 10
b = 20

The values of a and b were swapped inside the calling function but their values were intact after going back to the main function. This is because the variables a and b were sent as "Call by Value" arguments. And x and y in the called function are nothing but copies of a and b respectively.

Using Call by Reference :

```
int main()
{
        int a,b;
        a = 10;
        b = 20;
        Swap(&a,&b);
        printf("a = %d\nb = %d\n",a,b);
}

Swap(int *x,int *y)
{
        int z;
        z = *x;
        *x = * y;
        *y = z;
        printf("x = %d\ny = %d\n",*x,*y);

}
```

this function will give the following output :
x = 20
y = 10
a = 20
b = 10

The values of a and b were swapped inside the calling function using pointer syntax.Even after going back to main, the swapped values were retained. This is because the variables a and b were sent as "Call by Reference" arguments. And x and y in the called function are addresses of a and b respectively, thats why when we manipulated *x and *y , the values of a and b were also effected.

5) Arrays and Pointers

When an array is declared, the compiler allocates a base address and sufficient amount of storage to

contain all the elements of the array, in contiguous memory locations.
The compiler also defines the array name as a CONSTANT pointer to the first element.

Suppose we declare an array as follows :
static int x[5] = {1,2,3,4,5};

Suppose the base address of x is 1000 and assuming that each integer requires 4 bytes, the five elements will be stored at the following locations respectively : x[0] – 1000, x[1] – 1004, x[2] – 1008, x[3] = 1012, x[4] = 1016.

x = &x[0] = 1000

If we declare pointer p as an integer pointer, we can make the pointer to point to the array x by the following assignment : p = x;

this is equivalent to p = &x[0].

We can access every value of x, using p++ to move from one element to another. We cannot do the same with x, as it is created as a constant pointer by the compiler.

So the address of an array element can be calculated using its index and the scale factor of the data type.

Address of x[3] = base address + 3 * (scale factor of int)
= 1000 + (3 * 4) = 1012

p = &x[0] = 1000
p + 1 = &x[1]= 1004
p + 2 = &x[2] = 1008
p + 3 = &x[3] = 1012
p + 4 = &x[4] = 1016

Pointers can be used to manipulate two-dimentional arrays as well.

In a one dimentional array, the element x[i] is equivalent to *(p+i), where p = x.
Similarly in a two dimentional array, the element x[i][j] can be written as *((p+ i) + j) where p = x;

4) void *,malloc, calloc, realloc ,free(stdlib.h)

The process of allocating memory during program execution is called dynamic memory allocation.

C language offers 4 dynamic memory allocation functions. They are,
1.malloc()
2.calloc()
3.realloc()
4.free()

S.no	Function	Syntax
1	**malloc**	malloc (number *sizeof(int));

	()	
2	**calloc ()**	calloc (number, sizeof(int));
3	**realloc ()**	realloc (pointer_name, number * sizeof(int));
4	**free ()**	free (pointer_name);

malloc ():

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

Calloc ():

•calloc () function is also like malloc () function. But calloc () initializes the allocated memory to zero. But, malloc() doesn't.

Realloc ():

•realloc () function modifies the allocated memory size by malloc () and calloc () functions to new size.
•If enough space doesn't exist in memory of current block to extend, new block is allocated for the full size of reallocation, then copies the existing data to new block and then frees the old block.

free ():

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

Always check to see if the allocation request failed or succeeded while using dynamic memory allocation methods. Otherwise it may lead to NULL pointer access error.

6) Advantages and Disadvantages (sharing, dynamic alloc, errors etc)

Advantages :

- Pointers make it possible to share data among different sections of a program.
- Using pointers we can allocate memory dynamically and increase or decrease the amount of space allocated at runtime, unlike static arrays, for which the size has to be mentioned at the time of declaration.
- Function cannot return more than one value. But the same function can modify many pointer variables and function as if it is returning more than one variable.

Disadvantages :

1.) If sufficient memory is not available during runtime for the storage of pointers, the program may crash (least possible)
2.) If the programmer is not careful and consistent with the use of pointers, the program may crash (very possible)

7) Is it possible to have negative index in an array ? .

Yes. A negative index means the base address of the array is being decremented before dereferencing. The value at the resulting address will be accessed, though it can be a garbage value.

a[i] = *(a + i);
similarly a[-i] = *(a-i); which is completely valid for a pointer.

8) What is the format specifier for printing a pointer value ?

We can use %p or %x. %x will print address in hexa decimal format.

printf("x lives at %p.\n", (void*)&x);

other format specifiers :

%c – print a single character
%d – print a decimal integer
%e – print a floating point value in exponent form
%f – print a floating point value without exponent
%g – print a floating point value either e-type or f-type depending on value
%i – print a signed decimal integer
%o – print an octal integer, without leading zero
%s – print a string
%u – print an unsigned decimal integer
%x – print a hexadecimal integer, without leading 0x.
%p – prints the value of a pointer
%% - outputs a percent sign

9) What is the difference ?

Char a[] = “string”;
char *p = “literal”;
The first one a[] is a character array declaration. A is like any other array, we can safely modify its elements.

Second one is a pointer to a character array. 'p' is a static array of characters, we cannot modify p. The program might crash if p[i] is modified.

10) What will be the output of the below program ?

```
Int main()
{
        char *s = “Kernel Masters”;
        printf(“%d\t%d\n”,s[1],s[4]);
        printf(“%s\n”,s+s[1]-s[4]);
        printf(“%s\n”,s+s[3]-s[5]);

}
```

Output : 101 101
Kernel Masters
rnel Masters

Solution : Here the characters s[1] and s[4] are same. When printed with format specifier “%d” their ascii integer values are printed as 101.
Since both the values are same, s[1]-s[4] in second print statement will result in 0.
s + s[1] – s[4] = s;
Thus, the value at address s is printed as string, which is “Kernel Masters”.

Similarly, the difference between the ascii integer values of s[3] ('n') and s[5] ('l') will decide the output for second printf statement. Since both are adjacent characters as per ASCII, the difference might be 1 or 2 depending on the compiler. So the output will be printed accordingly.

11) Can we dereference a void pointer ?

A void pointer is a generalized pointer that doesn't specify which data type it is pointing to.
So, we cannot dereference a void pointer without typecasting it. The reason is, though the beginning address is available, since the compiler doesn't know which data type a pointer is pointing to, it will not know how many address locations it has to access to get the value. So we need to typecase it to let the compiler know of the scale factor.
Void * ptr;

printf (“%d\n”, *((int *)ptr));

12) What is the value of the expression 5[“Interview”];

it is same as “Interview”[5] = 'v';

we can interpret a[i] as *(a + i)
*(a + i) is equivalent to *(i + a) ,which in turn can be written as i[a].
so a[i] = i[a];

13)What is the output of the below statements:

int a = 1;
printf(“%d\t%d\t%d\t”,a,++a,a++);

C-s Calling convension is from right to left.

14) What is the output of the below statements
int i;
scanf(“%d”,i);
printf(“%d”,i);

Output : Error will be thrown as & symbol is missing before i. We have to specify the address of a variable to scanf, not the variable itself. However it will not throw compilation error, runtime error will come as the system is not able to access the memory location which the value of i.

15) Predict the output :
int main()

```
{
        int rk;

        printf(“%d”,scanf(“%d”,&rk));
}
```

Output : 1

the scanf function returns the number of arguments successfully read from the console. So the result here will be a 1.

16) What is the output

```
 void main()
{
printf(“sizeof(void *) = %d \n”, sizeof(void *));
printf(“sizeof(int *) = %d \n”,sizeof(int *));
printf(“sizeof(double *) =%d \n”, sizeof(double *));
printf(sizeof(struct unknown *) = %d \n”, sieof(struct unknown *));
}
```

Depending on the compiler, the output will be the size of an unsigned integer as the address locations are unsigned integers. Any pointer value will be an unsigned integer only ,so irrespective of the data type of pointer, the above statements will print 8 (4 in some compilers).