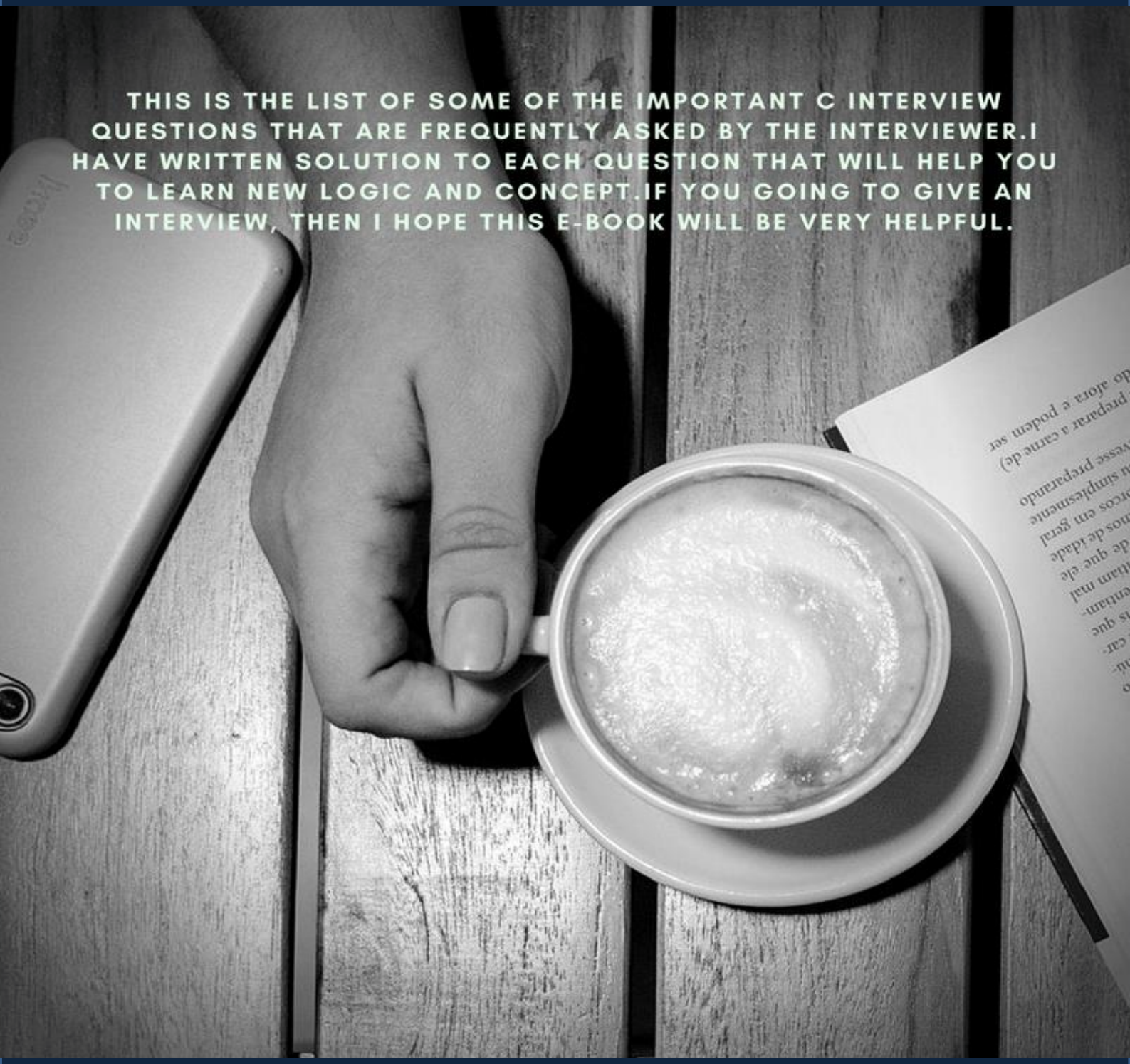
Aticleworld

100 C Interview Questions

www.aticleworld.com

Overview



Aticleworld
www.aticleworld.com

Question 1: What is the difference between declaration and definition of a variable?

Declaration of a variable in c

A variable declaration only provides sureness to the compiler at the compile time that variable exists with the given type and name, so that compiler proceeds for further compilation without needing all detail of this variable. In C language, when we declare a variable, then we only give the information to the compiler, but there is no memory reserve for it. It is only a reference, through which we only assure the compiler that this variable may be defined within the function or outside of the function.

Note: We can declare a variable multiple time but defined only once. eg,

```
extern int data;
extern int foo(int, int);
int fun(int, char); // extern can be omitted for function declarations
```

Definition of a variable in c

The definition is action to allocate storage to the variable. In another word, we can say that variable definition is the way to say the compiler where and how much to create the storage for the variable generally definition and declaration occur at the same time but not almost. **eg,**

```
int data;
int foo(int, int) { }
```

Note: When you define a variable then there is no need to declare it but vice versa is not applicable.

Question 2: What is the variable in C?

A variable defines a location name where we can put the value and we can use this value whenever required in the program. In another word, we can say that variable is a name (or identifier) which indicate some physical address in the memory, where data will be stored in form of the bits of string.
In C language, every variable has a specific data types (pre-defined or user-defined) that determine the size and memory layout of the variable.

Note: Each variable bind with two important properties, scope, and extent.

Question 3: Using the variable p write down some declaration

1. An integer variable.
2. An array of five integers.
3. A pointer to an integer.
4. An array of ten pointers to integers.
5. A pointer to a pointer to an integer.
6. A pointer to an array of three integers.
7. A pointer to a function that takes a pointer to a character as an argument and returns an integer.
8. An array of five pointers to functions that take an integer argument and return an integer.

Answer:

1. int p; // An integer
2. int p[5]; // An array of 5 integers
3. int *p; // A pointer to an integer
4. int *p[10]; // An array of 10 pointers to integers
5. int **p; // A pointer to a pointer to an integer
6. int (*p)[3]; // A pointer to an array of 3 integers
7. int (*p)(char *); // A pointer to a function a that takes an integer
8. int (*p[5])(int); // An array of 5 pointers to functions that take an integer argument and return an integer

Question 4: Some Questions related to declaration for you

1. int* (*fpData)(int , char, int (*paIndex)[3]);
2. int* (*fpData)(int , int (*paIndex)[3] , int (* fpMsg) (const char *));
3. int* (*fpData)(int (*paIndex)[3] , int (* fpMsg) (const char *), int (* fpCalculation[3]) (const char *));
4. int* (*fpData[2])(int (*paIndex)[3] , int (* fpMsg) (const char *), int (* fpCalculation[3]) (const char *));
5. int* (*(*fpData)(const char *))(int (*paIndex)[3] , int (* fpMsg) (const char *), int (* fpCalculation[3]) (const char *));

Question 5: What are storage classes in C language ?

The storage classes decide the extent (lifetime) and scope (visibility) of a variable or function within the program. Every variable gets some location in the memory where variable's value is stored in form of bits. Storage class defines the storage location of the variable like CPU register or memory (like the stack, BSS, DS).

There are four different storage class in C.

1. auto **2.** static **3.** extern **4.** register

Question 6: What are the data types in C?

There are two types of data type, user-defined and predefined. A predefined data type is int, char, float, double etc and user-defined is created by the users using the tags struct, union or enum. Basically, data types describe the size and memory layout of the variable.

In C language, different data types have the different ranges. The range varies from compiler to compiler. In below table, I have listed some data types with there ranges and format specifier as per the 32-bit GCC compiler.

Data Type	Memory (bytes)	Range	Format Specifier
short int	2	-32,768 to 32,767	%hd
unsigned short int	2	0 to 65,535	%hu
unsigned int	4	0 to 4,294,967,295	%u
int	4	-2,147,483,648 to 2,147,483,647	%d
long int	4	-2,147,483,648 to 2,147,483,647	%ld
unsigned long int	4	0 to 4,294,967,295	%lu
long long int	8	-(2^63) to (2^63)-1	%lld
unsigned long long int	8	0 to 18,446,744,073,709,551,615	%llu
signed char	1	-128 to 127	%c
unsigned char	1	0 to 255	%c
float	4	-	%f
double	8	-	%lf
long double	12	-	%Lf

Question 7: What are the uses of the keyword static?

In C language, the static keyword has a lot of importance. If we have used the static keyword with a variable or function, then only internal or none linkage is worked. I have described some simple use of a static keyword.

- A static variable only initializes once, so a variable declared static within the body of a function maintains its prior value between function invocations.
- A global variable with static keyword has an internal linkage, so it only accesses within the translation unit (.c). It is not accessible by another translation unit. The static keyword protects your variable to access from another translation unit.
- By default in C language, linkage of the function is external that it means it is accessible by the same or another translation unit. With the help of the static keyword, we can make the scope of the function local, it only accesses by the translation unit within it is declared.

Question 8: What is the difference between global and static global variables?

In simple word, they have different linkage.

- A static global variable ===>>> internal linkage.
- A non-static global variable ===>>> external linkage.

So global variable can be accessed outside of the file but the static global variable only accesses within the file in which it is declared.

Question 9: Differentiate between an internal static and external static variable?

In C language, the external static variable has the internal linkage and internal static variable has no linkage. So the life of both variable throughout the program but scope will be different.

- A external static variable ===>>> internal linkage.
- A internal static variable ===>>> none .

Question 10: What are the different types of linkage?

The C language has three types of linkage, **external linkage**, **internal linkage** and **none linkage**.

Question 11: Can static variables be declared in a header file?

Yes, we can declare the static variables in a header file.

Question 12: Size of the integer depends on what?

The C standard is explained that the minimum size of the integer should be 16 bits. Some programing language is explained that the size of the integer is implementation dependent but portable programs shouldn't depend on it.

Primarily size of integer depends on the type of the compiler which has written by compiler writer for the underlying processor. You can see compilers merrily changing the size of integer according to convenience and underlying architectures. So it is my recommendation use the C99 integer data types (uin8_t, uin16_t, uin32_t ..) in place of standard int.

Question 13: Are integers signed or unsigned?

According to C standard, integer data type is by default signed. So if you create an integer variable, it can store both positive and negative value.

Question 14: What is a difference between unsigned int and signed int in C?

The signed and unsigned integer type has the same storage (according to the standard at least 16 bits) and alignment but still, there is a lot of difference them, in bellows lines, I am describing some difference between the signed and unsigned integer.

- A signed integer can store the positive and negative value both but beside it unsigned integer can only store the positive value.
- The range of nonnegative values of a signed integer type is a sub-range of the corresponding unsigned integer type.
  For example,
  Assuming size of the integer is 2 bytes.
  signed int -32768 to +32767
  unsigned int 0 to 65535
- When computing the unsigned integer, it never gets overflow because if the computation result is greater than the largest value of the unsigned integer type, it is reduced modulo the number that is one greater than the largest value that can be represented by the resulting type.

  For example,
  Computational Result % (Largest value of the unsigned integer+1)

- The overflow of signed integer type is undefined.
- If Data is signed type negative value, the right shifting operation of Data is implementation dependent but for the unsigned type, it would be Data/ 2pos.
- If Data is signed type negative value, the left shifting operation of Data show the undefined behavior but for the unsigned type, it would be Data x 2pos.

Question 15: What is the difference between const and macro?

1. The const keyword is handled by the compiler, in another hand, a macro is handled by the preprocessor directive.
2. const is a qualifier that is modified the behavior of the identifier but macro is preprocessor directive.
3. There is type checking is occurred with const keyword but does not occur with #define.
4. const is scoped by C block, #define applies to a file.
5. const can be passed as a parameter (as a pointer) to the function. In case of call by reference, it prevents to modify the passed object value.

Question 16: What is the volatile keyword?

The volatile keyword is a type qualifier that prevents the objects from the compiler optimization. According to C standard, an object that has volatile-qualified type may be modified in ways unknown to the implementation or have other unknown side effects. You can also say that the value of the volatile-qualified object can be changed at any time without any action being taken by the code. If an object is qualified by the volatile qualifier, the compiler reloads the value from memory each time it is accessed by the program that means it prevents from to cache a variable into a register. Reading the value from the memory is the only way to check the unpredictable change of the value.

Question 17: Can we have a volatile pointer?

Yes, we can create a volatile pointer in C language.

int * volatile piData; // piData is a volatile pointer to an integer.

Question 18: What is the difference between a macro and a function?

Macro and function both are different things but still some times we can use macro in place of function. I have pointed some important difference between the macro and function in below table.

Macro	Function
There is no type checking.	Type checking is occurred.
Macro is Pre-processed	Function is Compiled.
Code Length Increases, when you call macro multiple times.	Code Length remains the same in every calling of the function.
Use of macro can lead Side effect.	No side Effect
Speed of Execution is Faster.	Speed of Execution is Slower as compare to macro.
Generally macro is useful for the small code.	Function is generally useful for the large code.
Because macro is pre- processed, so it is difficult to debug the macro.	Easy to debug the function.

Question 19: What is the difference between typedef & Macros?

typedef:

The C language provides a very important keyword typedef for defining a new name for existing types. The typedef is the compiler directive mainly use with user-defined data types (structure, union or enum) to reduce their complexity and increase the code readability and portability.

Syntax:
typedef type NewTypeName;

Let's take an example,

typedef unsigned int UnsignedInt;

Now UnsignedInt is a new type and using it, we can create a variable of unsigned int.

UnsignedInt Mydata;

In above example, Mydata is variable of unsigned int.

Note: A typedef creates synonyms or a new name for existing types it does not create new types.

Macro:

A macro is a pre-processor directive and it replaces the value before compiling the code. One of the major problem with the macro that there is no type checking. Generally, the macro is used to create the alias, in C language macro is also used as a file guard.

Syntax,

#define Value 10

Now Value becomes 10, in your program, you can use the Value in place of the 10.

Question 20: What do you mean by enumeration in C?

In C language enum is user-defined data type and it consists a set of named constant integer. Using the enum keyword, we can declare an enumeration type by using the enumeration tag (optional) and a list of named integer. An enumeration increases the readability of the code and easy to debug in comparison of symbolic constant (macro).The most important thing about the enum is that it follows the scope rule and compiler automatic assign the value to its member constant.

Note: A variable of enumeration type stores one of the values of the enumeration list defined by that type.

Syntax of enum,

enum Enumeration_Tag { Enumeration_List };

The Enumeration_Tag specifies the enumeration type name.

The Enumeration_List is a comma-separated list of named constant.

Example,

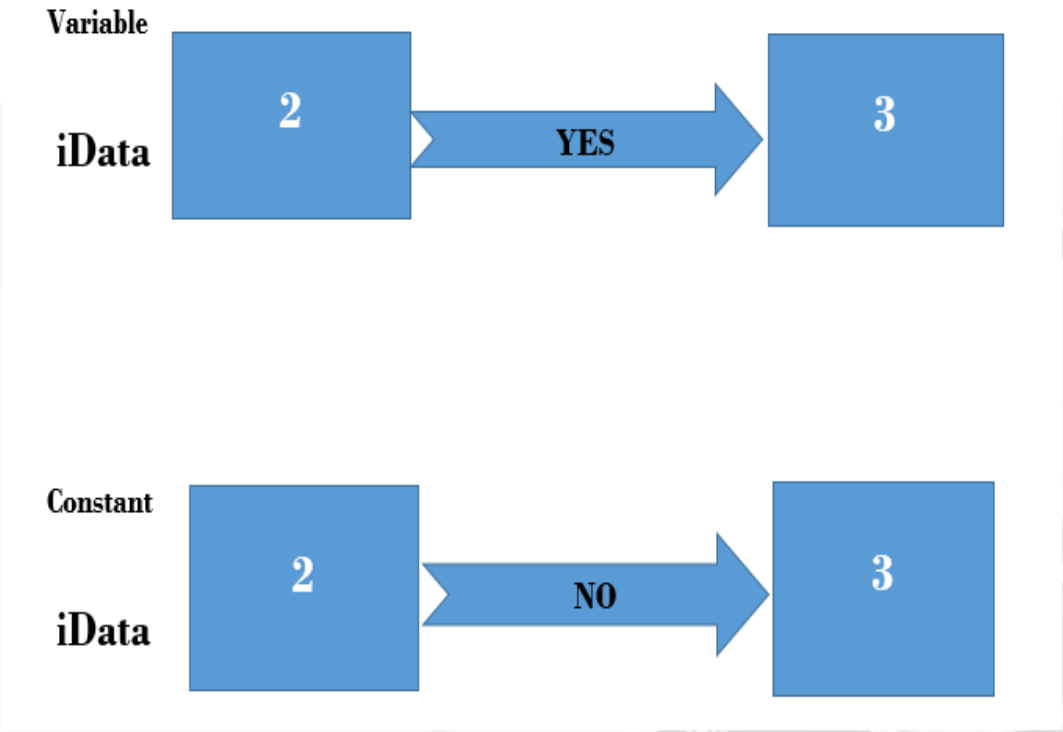
enum FLASH_ERROR { DEFRAGMENT_ERROR, BUS_ERROR};

Question 21: What does the keyword const mean?

A const is only a qualifier, it changes the behavior of a variable and makes it read-only type. When we want to make an object read-only type, then we have to declare it as const.

Syntax,

const DataType Identifier = Value;
e.g.
const int iData = 0;



At the time of declaration, const qualifier only gives the direction to the compiler that the value of declaring object could not be changed. In simple word, const means not modifiable (cannot assign any value to the object at the runtime).

Question 22: When should we use const in a C program?

There are following places where we need to use the const keyword in the programs.

- In call by reference function argument, if you don't want to change the actual value which has passed in function.
  E.g.
  int PrintData (const char *pcMessage);
- In some places, const is better then macro because const handle by the compiler and have a type checking.
  E.g.
  const int ciData = 100;
- In the case of I/O and memory mapped register const is used with the volatile qualifier for efficient access.
  E.g.
  const volatile uint32_t *DEVICE_STATUS = (uint32_t *) 0x80102040;
- When you don't want to change the value of an initialized variable.

Question 23: Differentiate between a constant pointer and pointer to a constant?

Constant pointer:

A constant pointer is a pointer whose value (pointed address) is not modifiable. If you will try to modify the pointer value, you will get the compiler error.

A constant pointer is declared as follows :

Data_Type * const Pointer_Name;

Let's see the below example code when you will compile the below code get the compiler error.

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

2. int main(void)
3. {
4.     int var1 = 10, var2 = 20;
5.
6.     //Initialize the pointer
7.     int *const ptr = &var1;
8.
9.     //Try to modify the pointer value
10.    ptr = &var2;
11.    printf("%d\n", *ptr);
12.
13.     return 0;
14.}
```

Pointer to a constant:

In this scenario the value of pointed address is constant that means we can not change the value of the address that is pointed by the pointer.

A pointer to a constant is declared as follows :

Data_Type const* Pointer_Name;

Let's take a small code to illustrate a pointer to a constant:

```
1. #include<stdio.h>
2.
3. int main(void)
4. {
5.     int var1 = 100;
6.     // pointer to constant integer
7.     const int* ptr = &var1;
8.
9.     //try to modify the value of pointed address
10.    *ptr = 10;
11.
12.    printf("%d\n", *ptr);
13.
14.    return 0;
15.}
```

Question 24: Which one is better: Pre-increment or Post increment?

Nowadays compiler is enough smart, they optimize the code as per the requirements. The post and pre increment both have own importance we need to use them as per the requirements.

If you are reading a flash memory byte by bytes through the character pointer then here you have to use the post-increment, either you will skip the first byte of the data. Because we already know that in case of pre-increment pointing address will be increment first and after that, you will read the value.

Let's take an example of the better understanding,

In below example code, I am creating a character array and using the character pointer I want to read the value of the array. But what will happen if I used pre-increment operator? The answer to this question is that 'A' will be skipped and B will be printed.

```
1. #include <stdio.h>
2.
3. int main(void)
4. {
5.
6.  char acData[5] ={'A','B','C','D','E'};
7.  char *pcData = NULL;
8.
9.  pcData = acData;
10.
11. printf("%c ",*++pcData);
12.
13. return 0;
14.}
```

But in place of pre-increment if we use post-increment then the problem is getting solved and you will get A as the output.

```
1. #include <stdio.h>
2.
3. int main(void)
4. {
5.
6.  char acData[5] ={'A','B','C','D','E'};
7.  char *pcData = NULL;
8.
9.  pcData = acData;
10.
11. printf("%c ",*pcData++);
12.
13. return 0;
14.}
```

Besides that, when we need a loop or just only need to increment the operand then pre-increment is far better than post-increment because in case of post increment compiler may have created a copy of old data which takes extra time. This is not 100% true because nowadays compiler is so smart and they are optimizing the code in a way that makes no difference between pre and post-increment. So it is my advice, if post-increment is not necessary then you have to use the pre-increment.

Note: Generally post-increment is used with array subscript and pointers to read the data, otherwise if not necessary then use pre in place of post-increment. Some compiler also mentioned that to avoid to use post-increment in looping condition.

```
1. iLoop = 0.

2. while (a[iLoop ++] != 0)
3. {
4. // Body statements
5. }
```

Question 25: Are the expressions *++ptr and ++*ptr same ?

Both expressions are different. Let's see a sample code to understand the difference between both expressions.

```
1. #include <stdio.h>
2.
3. int main(void)
4. {
5.     int aiData[5] = {100,200,300,400,500};
6.
7.     int *piData = aiData;
8.
9.     ++*piData;
10.
11.    printf("aiData[0] = %d, aiData[1] = %d, *piData = %d", aiData[0], aiData[1],
   *piData);
12.
13.    return 0;
14.}
15.
```

Output: 101 , 200 , 101

Explanation:

In the above example, two operators are involved and both have the same precedence with a right to left associativity. So the above expression ++*p is equivalent to ++ (*p). In another word, we can say it is pre-increment of value and output is 101, 200, 101.

```
1. #include <stdio.h>
2.
3. int main(void)
4. {
5.     int aiData[5] = {100,200,30,40,50};
6.
7.     int *piData = aiData;
8.
9.     *++piData;
10.
11.     printf("aiData[0] = %d, aiData[1] = %d, *piData = %d", aiData[0], aiData[1],
   *piData);
12.
13.    return 0;
14.}
```

Output: 100, 200, 200

Explanation:

In the above example, two operators are involved and both have the same precedence with the right to left associativity. So the above expression *++p is equivalent to *(++p). In another word you can say it is pre-increment of address and output is 100, 200,200.

Question 26: The Proper place to use the volatile keyword?

Accessing the memory-mapped peripherals register or hardware status register.

```
1. #define COM_STATUS_BIT  0x00000006

2. uint32_t const volatile * const pStatusReg = (uint32_t*)0x00020000;

3. unit32_t GetRecvData()
4. {
5.     //Code to recv data
6.     while (((*pStatusReg)  & COM_STATUS_BIT) == 0)
7.          {
8.         // Wait untill flag does not set
9.     }
10. return RecvData;
11.}
```

Sharing the global variables or buffers between the multiple threads.

Accessing the global variables in an interrupt routine or signal handler.

```
1. volatile int giFlag = 0;

2. ISR(void)
3. {
4.   giFlag = 1;
5. }

6. int main(void)
7. {
8.  while (!giFlag)
9.   {
10.     //do some work
11.  }
12.  return 0;
13.}
```

Question 27: Can a variable be both constant and volatile in C?

Yes, we can use both constant and volatile together. One of the great use of volatile and const together at the time of accessing the GPIO registers. In case of GPIO, its value can be changed by the 'external factors' (if a switch or any output device is attached with GPIO), if it is configured as an input. In that situation, volatile plays an important role and ensures that the compiler always read the value from the GPIO address and avoid to make any assumption.

After using the volatile keyword, you will get the proper value whenever you are accessing the ports but still here is one more problem because the pointer is not const type so it might be your program change the pointing address of the pointer. So we have to create a constant pointer with volatile keyword.

Syntax of declaration,

int volatile * const PortRegister;

How to read the above declaration,

```
int volatile * const PortRegister;
 |     |     |   |    |
 |     |     |   |    +------> PortRegister is a
 |     |     |   +-----------> constant
 |     |     +---------------> pointer to a
 |     +---------------------> volatile
 +-------------------------> integer
```

Consider a simple below example:

#define PORTX 0x00020000 // Address of the GPIO

uint32_t volatile * const pcPortReg = (uint32_t *) PORTX;

The pcPortReg is a constant pointer to a volatile unsigned integer, using *pcPortReg we can access the memory-mapped register.

*pcPortReg = value; // Write value to the port

value = *pcPortReg; // Read value from the port

Question 28: What is the difference between the const and volatile qualifier in C?

The const keyword is compiler-enforced and says that program could not change the value of the object that means it makes the object nonmodifiable type.

e.g.,
const int a = 0;

if you will try to modify the value of "a", you will get the compiler error because "a" is qualified with const keyword that prevents to change the value of the integer variable.

In another side volatile prevent from any compiler optimization and says that the value of the object can be changed by something that is beyond the control of the program and so that compiler will not make any assumption about the object.

e.g.,
volatile int a;

When the compiler sees the above declaration then it avoids to make any assumption regarding the "a" and in every iteration read the value from the address which is assigned to the variable.

Question 29: How to set, clear, toggle and checking a single bit in C?

Setting a Bits

Bitwise OR operator (|) use to set a bit of integral data type. “OR” of two bits is always one if any one of them is one.

Number | = (1<< nth Position)

Clearing a Bits

Bitwise AND operator (&) use to clear a bit of integral data type. “AND” of two bits is always zero if any one of them is zero. To clear the nth bit, first, you need to invert the string of bits then AND it with the number.

Number &= ~ (1<< nth Position)

Checking a Bits

To check the nth bit, shift the ‘1’ nth position toward the left and then “AND” it with the number.

Bit = Number & (1 << nth)

Toggling a Bits

Bitwise XOR (^) operator use to toggle the bit of an integral data type. To toggle the nth bit shift the ‘1’ nth position toward the left and “XOR” it.

Number ^= (1<< nth Position)

Question 30: How to print a decimal number in binary format in C?

```
1. #define CHAR_BITS  8  // size of character
2. #define INT_BITS  ( sizeof(int) * CHAR_BITS) //bits in integer
3.
4.
5. void PrintInBinary(unsigned n)
6. {
7.     char Pos = (INT_BITS -1);
8.
9.     for (; Pos >= 0 ; --Pos)
10.    {
11.       (n & (1 << Pos))? printf("1"): printf("0");
12.    }
13.
14.}
```

Question 31: Write an Efficient C Program to Reverse Bits of a Number?

It is a simple algorithm to reverse bits of the 32-bit integer. This algorithm uses the eight constant value for the reversing the bits and takes five simple steps.

In below section, I am describing the functioning of each step.

Steps 1:

num = (((num & 0xaaaaaaaa) >> 1) | ((num & 0x55555555) << 1));

This expression used to swap the bits, suppose num is 0100, after the above expression it will be 1000.

Steps 2:

num = (((num & 0xcccccccc) >> 2) | ((num & 0x33333333) << 2));

Above expression uses to swap the 2 bits of a nibble. Suppose num is 10 00, after the above expression, it will be 00 01.

Steps 3:

num = (((num & 0xf0f0f0f0) >> 4) | ((num & 0x0f0f0f0f) << 4));

An expression used to swaps the nibbles. like if num is 0011 0010 then after the above expression it will be 0010 0011.

Steps 4:

num = (((num & 0xff00ff00) >> 8) | ((num & 0x00ff00ff) << 8));

This statement uses to swap the bytes of an integer. Let num is 00001000 00001100, after the above expression, it will be 00001100 00001000.

Steps 5:

((num >> 16) | (num << 16));

The above expression uses to swap the half-word of an integer. Means that if the num is 0000000011001110 1000100100000110 after the above result number will be 1000100100000110 0000000011001110.

//bit reversal function

```
1. unsigned int ReverseTheBits(register unsigned int x)
2. {
3.   x = (((x & 0xaaaaaaaa) >> 1) | ((x & 0x55555555) << 1));
4.   x = (((x & 0xcccccccc) >> 2) | ((x & 0x33333333) << 2));
5.   x = (((x & 0xf0f0f0f0) >> 4) | ((x & 0x0f0f0f0f) << 4));
6.   x = (((x & 0xff00ff00) >> 8) | ((x & 0x00ff00ff) << 8));
7.
8.   return((x >> 16) | (x << 16));
9.
10.}
```

Question 32: What is the output of the below program?

```
1. #include <stdio.h>
2.
3. int main()
4. {
5.       int data = 16;
6.
7.       data = data >> 1;
8.
9.       printf("%d\n", data );
10.}
```

Output: 8

Question 33: What is the output of the below program?

```
1. #include <stdio.h>
2.
3. int main()
4. {
5.   int c = 8 ^7;
6.
7.   printf("%d\n", c);
8. }
```

Output: 15

Question 34: What is the output of the below program?

```
1. #include <stdio.h>
2. #include<stdlib.h>
3.
4. int main()
5. {
6.    void *pvBuffer = NULL;
7.
8.    pvBuffer = malloc(sizeof(int));
9.
10.   *((int*)pvBuffer) = 0x00000000;
11.
12.   *((int*)pvBuffer)|= 2;
13.
14.   printf("OutPut = %d",*((int*)pvBuffer));
15.
16.   free(pvBuffer);
17.
18.}
```

Output: 15

Question 36: Write a program swap two numbers without using the third variable?

Let's assume a, b two numbers, there are a lot of methods two swap two number without using the third variable.

Method 1((Using Arithmetic Operators):

```
1. #include <stdio.h>
2.
3. int main()
4. {
5.   int a = 10, b = 5;
6.
7.   // algo to swap 'a' and 'b'
8.   a = a + b;  // a becomes 15
9.   b = a - b;  // b becomes 10
10.  a = a - b;  // fonally a becomes 5
11.
12.  printf("After Swapping the value of: a = %d, b = %d\n\n", a, b);
13.
14.  return 0;
15.}
```

Method 2 (Using Bitwise XOR Operator):

```
1. #include <stdio.h>
2.
3. int main()
4. {
5.   int a = 10, b = 5;
6.
7.   // algo to swap 'a' and 'b'
8.   a = a ^ b;  // a becomes (a ^ b)
9.   b = a ^ b;  // b = (a ^ b ^ b), b becomes a
10.  a = a ^ b;  // a = (a ^ b ^ a), a becomes b
11.
12.  printf("After Swapping the value of: a = %d, b = %d\n\n", a, b);
13.
14.  return 0;
15.}
```

Question 37: Write a program to check an integer is a power of 2?

Here, I am writing a small algorithm to check the power of 2. If a number is a power of 2, function return 1.

```
1. int CheckPowerOftwo (unsigned int x)
2. {
3.  return ((x != 0) && !(x & (x - 1)));
4. }
```

Question 38: What is the output of the below code?

```
1. #include <stdio.h>
2.
3.  int main()
4.  {
5.      int x = -30;
6.
7.      x = x << 1;
8.
9.      printf("%d\n", x);
10. }
```

Output: undefined behavior

Question 39: What is the output of the below code?

```
1. #include <stdio.h>
2.
3.  int main()
4.  {
5.       int x = -30;
6.
7.       x = x >> 1;
8.
9.       printf("%d\n", x);
10.}
```

Output: implementation-defined.

Question 40: Write a program to count set bits in an integer?

```
1. unsigned int NumberSetBits(unsigned int n)
2. {
3.   unsigned int CountSetBits= 0;
4.   while (n)
5.   {
6.     CountSetBits += n & 1;
7.     n >>= 1;
8.   }
9.   return CountSetBits;
10.}
```

Question 41: When should we use pointers in a C program?

- To pass large structure liked server request or response packet.
- To implement the linked list and binary trees.
- To play with GPIO or hardware register.
- To get the address or update value from the function (call by reference)
- To create the dynamic array.
- To create call back function using the function pointer.

Note: Besides it, lots of places where the need to use the pointer.

Question 42: What is void or generic pointers in C?

A void pointer is a generic pointer. It has no associated data type that's why it can store the address of any type of object and type-casted to any types.

According to C standard, the pointer to void shall have the same representation and alignment requirements as a pointer to a character type. A void pointer declaration is similar to the normal pointer, but the difference is that instead of data types we use the void keyword.

Syntax:

void * Pointer_Name;

Question 43: What is the advantage of a void pointer in C?

There are following advantages of a void pointer in c.

Using the void pointer we can create a generic function that can take arguments of any data type. The memcpy and memmove library function are the best examples of the generic function, using these function we can copy the data from the source to destination.

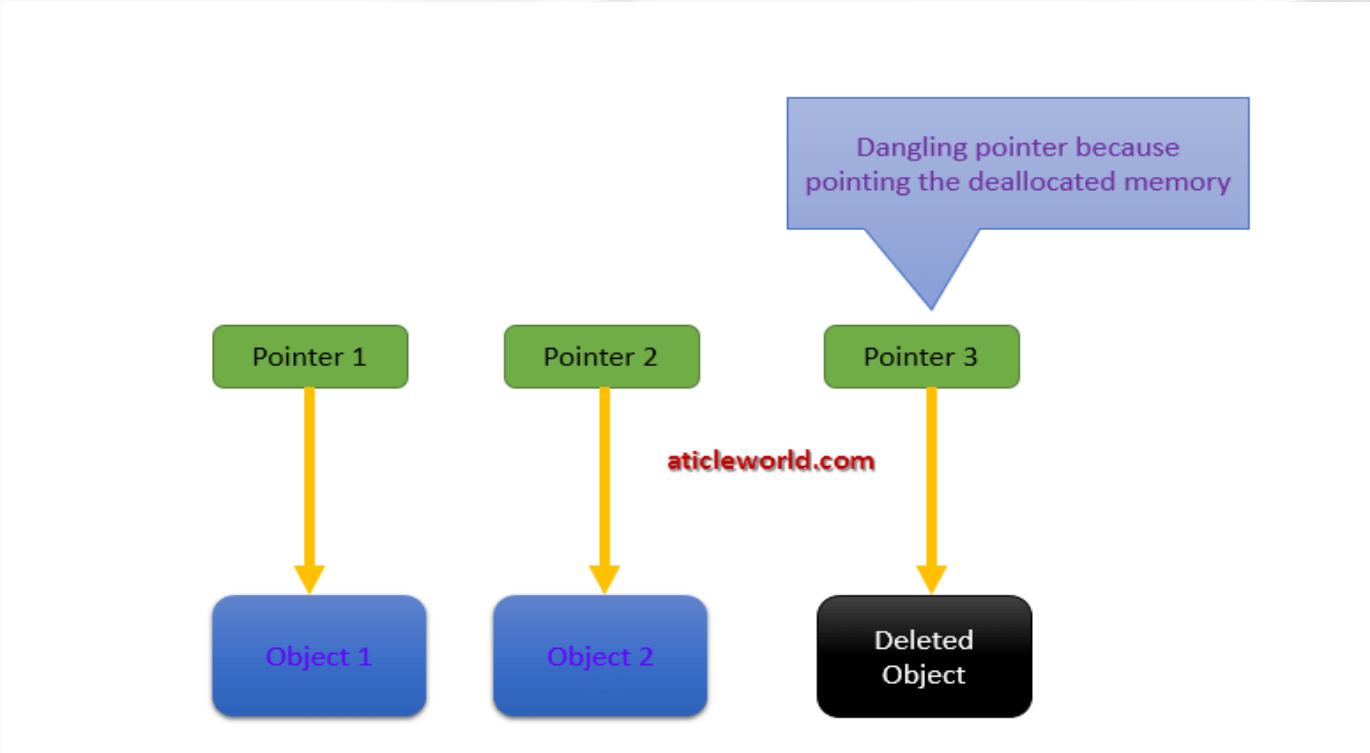
e.g.
void * memcpy (void * dst, const void * src, size_t num);

We have already know that void pointer can be converted to another data type that is the reason malloc, calloc or realloc library function return void *. Due to the void * these functions are used to allocate memory to any data type.

Using the void * we can create a generic linked.

Question 44: What are dangling pointers?

Generally, daggling pointers arise when the referencing object is deleted or deallocated, without changing the value of the pointers. It creates the problem because the pointer is still pointing the memory that is not available. When the user tries to dereference the daggling pointers than it shows the undefined behavior and can be the cause of the segmentation fault.



For example,

```
1. #include<stdio.h>
2. #include<stdlib.h>
3.
4. int main()
5. {
6. int *piData = NULL;
7.
8. piData = malloc(sizeof(int)* 10); //creating integer of size 10.
9.
10.free(piData); //free the allocated memory
11.
12.*piData = 10; //piData is dangling pointer
13.
14.return 0;
15.
16.}
```

In simple word, we can say that dangling pointer is a pointer that not pointing a valid object of the appropriate type and it can be the cause of the undefined behavior.

Question 45: What is the wild pointer?

A pointer that is not initialized properly prior to its first use is known as the wild pointer. Uninitialized pointers behavior is totally undefined because it may point some arbitrary location that can be the cause of the program crash, that's is the reason it is called a wild pointer.
In the other word, we can say every pointer in programming languages that are not initialized either by the compiler or programmer begins as a wild pointer.

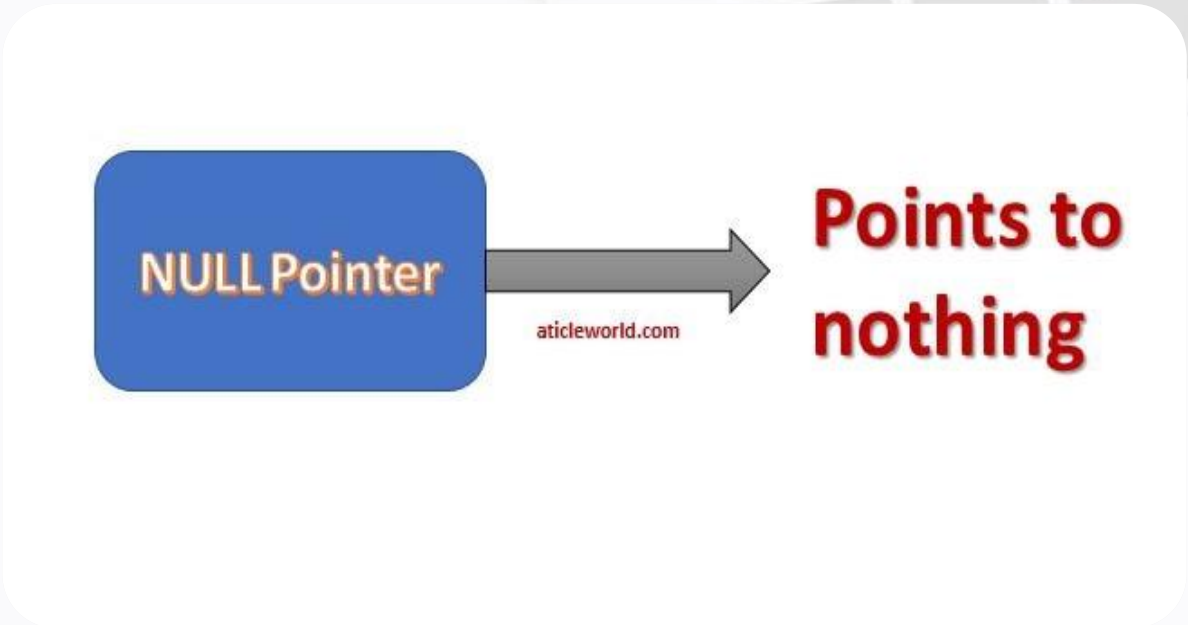
Note: Generally, compilers warn about the wild pointer.

Syntax,

```
int *piData; //piData is wild pointer
```

Question 46: What is a NULL pointer?

According to C standard, an integer constant expression with the value 0, or such an expression cast to type void *, is called a null pointer constant. If a null pointer constant is converted to a pointer type, the resulting pointer, called a null pointer.



Syntax,

```
int *piData = NULL; // piData is a null pointer
```

Question 47: What is a Function Pointer?

A function pointer is similar to the other pointers but the only difference is that it points to a function instead of the variable. In the other word, we can say, a function pointer is a type of pointer that store the address of a function and these pointed function can be invoked by function pointer in a program whenever required.

Question 48: How to declare a pointer to a function in c?

The syntax for declaring function pointer is very straightforward. It seems like difficult in beginning but once you are familiar with function pointer then it becomes easy.

The declaration of a pointer to a function is similar to the declaration of a function. That means function pointer also requires a return type, declaration name, and argument list. One thing that you need to remember here is, whenever you declare the function pointer in the program then declaration name is preceded by the * (Asterisk) symbol and enclosed in parenthesis.

For example,

```
void ( *fpData )( int );
```

For the better understanding, let's take an example to describe the declaration of a function pointer in c.
e.g.,

```
void ( *pfDisplayMessage) (const char *);
```

In above expression, pfDisplayMessage is a pointer to a function taking one argument, const char *, and returns void.

When we declare a pointer to function in c then there is a lot of importance of the bracket. If in the above example, I remove the bracket, then the meaning of the above expression will be change and it becomes void *pfDisplayMessage (const char *). It is a declaration of a function which takes the const character pointer as arguments and returns void pointer.

Question 49: Where can the function pointers be used?

There are a lot of places, where the function pointers can be used. Generally, function pointers are used in the implementation of the callback function, finite state machine and to provide the feature of polymorphism in C language ...etc.

Question 50: What is the difference between array and pointer in c?

Here is one important difference between array and pointer is that address of the element in an array is always fixed we cannot modify the address at execution time but in the case of pointer we can change the address of the pointer as per the requirement.

Consider the below example:

In below example when trying to increment the address of the array then we will get the compiler error.

```
1   #include <stdio.h>
2   #include <stdlib.h>
3
4
5   int main(int argc, char *argv[]) {
6
7    char acBuffer [] = {'a','t','i','c' ,'l' ,'e'}; // array of character
8
9    acBuffer++; // increment the array
10
11   //print the element
12   printf("Element of the array %d\n",*acBuffer);
13
14
15  return 0;
16  }
```

ces | Compile Log | Debug | Find Results | Close

	Message
HRA\Desktop\pointer\main.c	**In function 'main':**
RA\Desktop\pointer\main.c	[Error] lvalue required as increment operand
RA\Desktop\pointer\Makefile.win	recipe for target 'main.o' failed

Note: When an array is passed to a function then it decays its pointer to the first element.

Question 51: What is static memory allocation and dynamic memory allocation?

According to C standard, there are four storage duration, static, thread (C11), automatic, and allocated. The storage duration determines the lifetime of the object.

The static memory allocation:

Static Allocation means, an object has external or internal linkage or declared with static storage-class. It's initialized only once, prior to program startup and its lifetime is throughout the execution of the program. A global and static variable is an example of static memory allocation.

The dynamic memory allocation:

In C language, there are a lot of library functions (malloc, calloc, or realloc,..) which are used to allocate memory dynamically. One of the problems with dynamically allocated memory is that it is not destroyed by the compiler itself that means it is the responsibility of the user to deallocate the allocated memory.

When we allocate the memory using the memory management function, they return a pointer to the allocated memory block and the returned pointer is pointing to the beginning address of the memory block. If there is no space available, these functions return a null pointer.

Question 52: What is the memory leak in C?

A memory leak is a common and dangerous problem. It is a type of resource leak. In C language, a memory leak occurs when you allocate a block of memory using the memory management function and forget to release it.

```
1. int main ()
2. {
3.
4.    char * pBuffer = malloc(sizeof(char) * 20);
5.
6.    /* Do some work */
7.
8.    return 0; /*Not freeing the allocated memory*/
9. }
```

Note: once you allocate a memory than allocated memory does not allocate to another program or process until it gets free.

Question 53: What is the difference between malloc and calloc?

The malloc and calloc are memory management functions. They are used to allocate memory dynamically. Basically, there is no actual difference between calloc and malloc except that the memory that is allocated by calloc is initialized with 0.

In C language,calloc function initialize the all allocated space bits with zero but malloc does not initialize the allocated memory. These both function also has a difference regarding their number of arguments, malloc take one argument but calloc takes two.

Question 54: What is the return value of malloc (0)?

If the size of the requested space is zero, the behavior will be implementation-defined. The return value of the malloc could be a null pointer or it shows the behavior of that size is some nonzero value. It is suggested by the standard to not use the pointer to access an object that is returned by the malloc while size is zero.

Question 55: What is the purpose of realloc( )?

The realloc function is used to resize the allocated block of memory. It takes two arguments first one is a pointer to previously allocated memory and the second one is the newly requested size.

The realloc function first deallocates the old object and allocates again with newly specified size. If the new size is lesser to the old size, the contents of the newly allocated memory will be same as prior but if any bytes in the newly created object goes beyond the old size, the values of the exceeded size will be indeterminate.

Syntax:

void *realloc(void *ptr, size_t size);

Example,

```
1. #include <stdio.h>
2. #include <stdlib.h>
3. #include <string.h>
4.
5. int main ()
6. {

7. char *pcBuffer = NULL;

8. /* Initial memory allocation */
9. pcBuffer = malloc(8);

10.strcpy(pcBuffer, "aticle");
11.printf("pcBuffer = %s\n", pcBuffer);
12.
13./* Reallocating memory */
14.pcBuffer = realloc(pcBuffer, 15);
15.
16.strcat(pcBuffer, "world");
17.printf("pcBuffer = %s\n", pcBuffer);
18.
19.//free the allocated memory
20.free(pcBuffer);
21.
22.return 0;
23.}
```

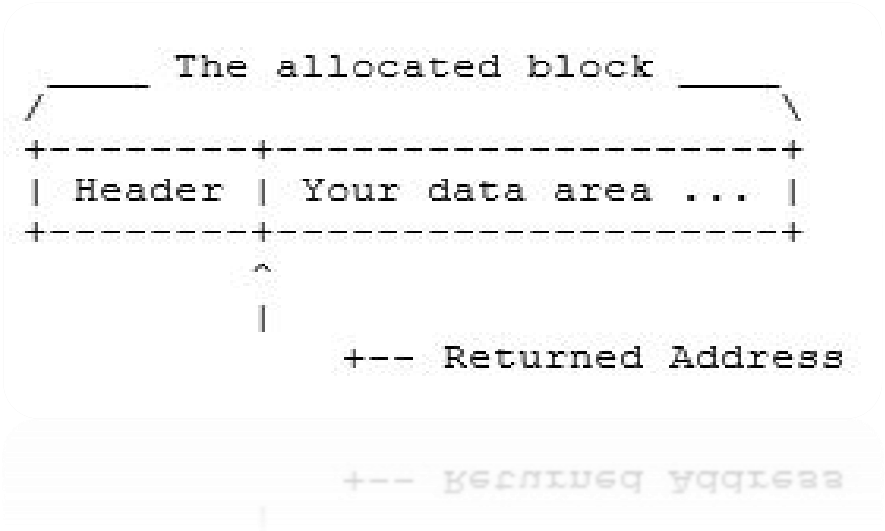
Output:

pcBuffer = aticle
pcBuffer = aticleworld

Question 56: How is the free work in C?

When we call the memory management functions (malloc, calloc or realloc) then these functions keep extra bytes for bookkeeping.

Whenever we call the free function and pass the pointer that is pointing to allocated memory, the free function gets the bookkeeping information and release the allocated memory. Anyhow if you or your program change the value of the pointer that is pointing to the allocated address, the calling of free function give the undefined result.



For example,

```
1. #include <stdio.h>
2. #include <stdlib.h>
3.
4.
5. int main()
6. {
7. char *pcBuffer = NULL;
8. pcBuffer  =  malloc(sizeof(char) *  16); //Allocate the memory
9.
10.pcBuffer++; //Increment the pointer
11.
12.free(pcBuffer); //Call free function to release the allocated memory
13.
14.return 0;
15.}
```

Question 57: What is the difference between memcpy and memmove?

Both copies function are used to copy n characters from the source object to destination object but they have some difference that is mentioned below.

- The memcpy copy function shows undefined behavior if the memory regions pointed to by the source and destination pointers overlap. The memmove function has the defined behavior in case of overlapping. So whenever in doubt, it is safer to use memmove in place of memcpy.

See the below code,

```
1. #include <string.h>
2. #include <stdio.h>
3.
4.
5. char str1[50] = "I am going from Delhi to Gorakhpur";
6. char str2[50] = "I am going from Gorakhpur to Delhi";
7.
8. int main( void )
9. {
10.
11.//Use of memmove
12.printf( "Function:\tmemmove with overlap\n" );
13.printf( "Orignal :\t%s\n",str1);
14.printf( "Source:\t\t%s\n", str1 + 5 );
15.printf( "Destination:\t%s\n", str1 + 11 );
16.memmove( str1 + 11, str1 + 5, 29 );
17.printf( "Result:\t\t%s\n", str1 );
18.printf( "Length:\t\t%d characters\n\n", strlen( str1 ) );
19.
20.//Use of memcpy
21.printf( "Function:\tmemcpy with overlap\n" );
22.printf( "Orignal :\t%s\n",str2);
23.printf( "Source:\t\t%s\n", str2 + 5 );
24.printf( "Destination:\t%s\n", str2 + 11 );
25.memcpy( str2 + 11, str2 + 5, 29 );
26.printf( "Result:\t\t%s\n", str2 );
27.printf( "Length:\t\t%d characters\n\n", strlen( str2 ) );
28.
29.return 0;
30.}
```

OutPut:

Function: memmove with overlap
Orignal : I am going from Delhi to Gorakhpur
Source: going from Delhi to Gorakhpur
Destination: from Delhi to Gorakhpur
Result: I am going going from Delhi to Gorakhpur
Length: 40 characters
Function: memcpy with overlap
Orignal : I am going from Gorakhpur to Delhi
Source: going from Gorakhpur to Delhi
Destination: from Gorakhpur to Delhi
Result: I am going going fring frakg frako frako
Length: 40 characters.

- The memmove function is slower in comparison to memcpy because in memmove extra temporary array is used to copy n characters from the source and after that, it uses to copy the stored characters to the destination memory.

- The memcpy is useful in forwarding copy but memmove is useful in case of overlapping scenario.

Question 58: Reverse a string in c without using library function.

A string is a collection of characters and it always terminates with a null character that means every string contains a null character at the end of the string.

Example:

char *pszData = “aticle”;

In above example, pszData is the pointer to the string. All characters of the string are stored in a contiguous memory and consist a null character in the last of the string.

See the below table,

character	‘a’	‘t’	‘i’	‘c’	‘l’	‘e’	‘\0’
Address	0x00	0x01	0x02	0x03	0x04	0x05	0x06

Reversing a string using the Iterative method:

Algorithm:

- Calculate the length (Len) of string.
- Initialize the indexes of the array.
  Start = 0, End = Len-1
- In a loop, swap the value of pszData[Start] with pszData[End].
- Change the indexes’ of the array as follows.
  Start = start +1; End = end – 1

Method 1:

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

3. int main()
4. {
5.     char acData[100]={0}, Temp = 0;
6.     int iLoop =0, iLen = 0;
7.     int cnt =0;
8.
9.     printf("\nEnter the string :");
10.
11.    if (fgets(acData,100,stdin) == NULL) //Maximum takes 100 character
12.    {
13.        printf("Error\n");
14.        return 1;
15.    }
16.
17.    while(acData[iLen++] != '\0'); //calculate length of string
18.
19.    iLen--;  //Remove the null character

20.    iLen--;  //Array index start from 0 to (length -1)

21.    while (iLoop < iLen) {
22.
23.      Temp = acData[iLoop];
24.       acData[iLoop] = acData[iLen];
25.       acData[iLen] = Temp;
26.       iLoop++;
27.       iLen--;
28.    }
29.
30.    printf("\n\nReverse string is : %s\n\n",acData);
31.    return 0;
32.}
```

Method 2:

```
1. #include <stdio.h>
2. #include <stdlib.h>
3.
4. #define SWAP_CHARACTER(a,b)  do { \
5.
                        (*a)^=(*b); \
6.
                        (*b)^=(*a);\
7.
                        (*a)^=(*b);\
8.
                        a++; \
9.
                        b--; \
10.
           }while(0);
11.
12.int main(int argc, char *argv[])
13.{
14.  char acData[100]={0};
15.  char *pcStart = NULL;
16.  char *pcEnd = NULL;
17.  int iLoop =0, iLen = 0;
18.
19.   printf("\nEnter the string :");
20.
21. //Maximum takes 100 character
22.   if (fgets(acData,100,stdin) == NULL)
23.   {
24.        printf("Error\n");
25.        return 1;
26.   }
27.   //Pointer point to the address of first character
28.   pcStart = acData;
29.
30.   // calculate length of string
31.   while(acData[iLen++] != '\0');
32.
33.  //Remove the null character
34.    iLen--;
35.
36.    pcEnd = (pcStart + iLen-1);
37.
38.   while (iLoop < iLen/2) {
39.
40.    SWAP_CHARACTER (pcStart,pcEnd);
41.      iLoop++;
42.
43.   }
44.
45.   printf("\n\nReverse string is : %s\n\n",acData);
46.   return 0;
47.}
```

Recursive way to reverse a string

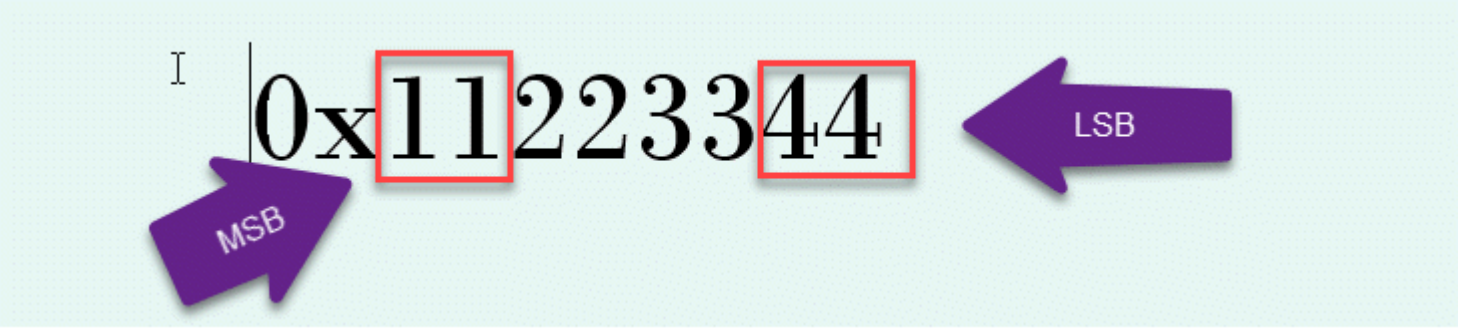
Algorithm:

- Calculate the length (Len) of string.
- Initialize the indexes of the array.
  Start = 0, End = Len-1
- swap the value of pszData[Start] with pszData[End].
- Change the indexes' of an array as below and Recursively call reverse function for the rest array.
  Start = start +1; End = end – 1

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

3. int StringRev(char *pszInputData, unsigned int Start, unsigned int End)
4. {
5.    if(Start >= End)
6.    {
7.           return 1;
8.    }
9.    // swap the data
10.   *(pszInputData + Start) = *(pszInputData + Start) ^ *(pszInputData + End);
11.   *(pszInputData + End) = *(pszInputData + Start) ^ *(pszInputData + End);
12.   *(pszInputData + Start) = *(pszInputData + Start) ^ *(pszInputData + End);
13.
14.   //function called repeatedly
15.    StringRev(pszInputData,Start+1, End-1);
16.
17. }
18. int main(int argc, char *argv[]) {
19.
20.  char acData[100]={0};
21.  int  iLen = 0;
22.  unsigned int Start=0;
23.
24.   printf("\nEnter the string :");
25.
26.   //Maximum takes 100 character
27.   if (fgets(acData,100,stdin) == NULL)
28.   {
29.        printf("Error\n");
30.        return 1;
31.   }
32.    while(acData[iLen++] != '\0'); // calculate length of string
33.
34.    iLen--; //Remove the null character
35.    iLen--; //Find array last index
36.
37.    StringRev(acData,Start, iLen);
38.
39.    printf("\n\nReverse string is : %s\n\n",acData);
40.
41.   return 0;
42.}
```

Question 59: What is big-endian and little-endian?

Suppose, 32 bits Data is 0x11223344.



Big-endian

The most significant byte of data stored at the lowest memory address.

Address	Value
00	0x11
01	0x22
02	0x33
03	0x44

little-endian.

The least significant byte of data stored at the lowest memory address.

Address	Value
00	0x44
01	0x33
02	0x22
03	0x11

Note: Some processor has the ability to switch one endianness to other endianness using the software means it can perform like both big endian or little endian at a time. This processor is known as the Bi-endian, here are some architecture (ARM version 3 and above, Alpha, SPARC) who provide the switchable endianness feature.

Question 60: Write a c program to check the endianness of the system.

Method 1

Using the pointers, we can check the endianness of the system. See the below example code,

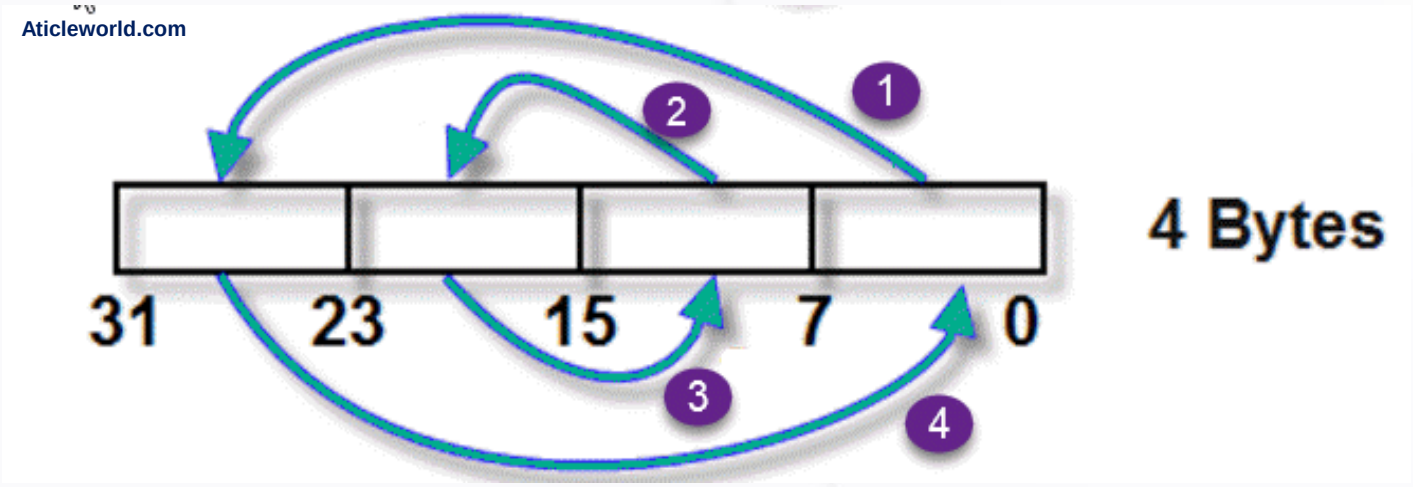
```
1. #include <stdio.h>
2. #include <stdlib.h>
3. #include <inttypes.h>
4.
5. int main(void)
6. {
7. uint32_t u32RawData;
8. uint8_t *pu8CheckData;
9. u32RawData = 0x11223344; //Assign data

10.
11.pu8CheckData = (uint8_t *)&u32RawData; //Type cast
12.
13.if (*pu8CheckData == 0x44) //check the value of lower address
14.{
15. printf("little-endian");
16.}
17.else if (*pu8CheckData == 0x11) //check the value of lower address
18.{
19. printf("big-endian");
20.}
21.
22.return 0;
23.}
```

Method 2

Using the union, we can also check the endianness of the system. See the below example code,

```
1. #include <stdio.h>
2. #include <stdlib.h>
3. #include <inttypes.h>
4.
5. typedef union
6. {
7.
8. uint32_t u32RawData;  // integer variable
9. uint8_t  au8DataBuff[4]; //array of character
10.
11.}RawData;
12.
13.int main(void)
14.{
15.RawData uCheckEndianess;
16.uCheckEndianess.u32RawData = 0x11223344; //assign the value
17.
18.if (uCheckEndianess.au8DataBuff[0] == 0x44) //check the array first index value
19.{
20.printf("little-endian");
21.}
22.else if (uCheckEndianess.au8DataBuff[0] == 0x11) //check the array first index value
23.{
24.printf("big-endian");
25.}
26.
27.return 0;
28.}
```

Question 61: How to convert little endian to big endian vice versa in C?



Method 1:

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

//Function to change the endianess
uint32_t ChangeEndianness(uint32_t u32Value)
{
uint32_t u32Result = 0;
u32Result |= (u32Value & 0x000000FF) << 24;
u32Result |= (u32Value & 0x0000FF00) << 8;
u32Result |= (u32Value & 0x00FF0000) >> 8;
u32Result |= (u32Value & 0xFF000000) >> 24;
return u32Result;
}


int main()
{
uint32_t u32CheckData  = 0x11223344;
uint32_t u32ResultData =0;
u32ResultData = ChangeEndianness(u32CheckData);  //swap the data
printf("0x%x\n",u32ResultData);
u32CheckData = u32ResultData;
u32ResultData = ChangeEndianness(u32CheckData);//again swap the data
printf("0x%x\n",u32ResultData);
return 0;
}
```

Method 2:

We can also make the macro to swap the data from one endianness to another.

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


//Macro to swap the byte
#define ChangeEndianness(A)   ((((uint32_t)(A) & 0xff000000) >> 24) | (((uint32_t)(A) &
0x00ff0000) >> 8)  | \
(((uint32_t)(A) & 0x0000ff00) << 8)  | (((uint32_t)(A) & 0x000000ff) << 24))
```

```
int main()
{
uint32_t u32CheckData  = 0x11223344;
uint32_t u32ResultData =0;

u32ResultData = ChangeEndianness(u32CheckData);
printf("0x%x\n",u32ResultData);

u32CheckData = u32ResultData;

u32ResultData = ChangeEndianness(u32CheckData);
printf("0x%x\n",u32ResultData);

return 0;
}
```

Question 62: How to find the size of an array in C without using the sizeof operator?

```
#include <stdio.h>

//Macro to calculate size
#define SIZEOF(Var) ((char*)(&Var + 1) -(char*)&Var)

int main(int argc, char *argv[])
{

  int iTotalElement = 0 ;

  int  aiData[] = {10, 20, 30, 40, 50, 60};

  iTotalElement = SIZEOF(aiData)/SIZEOF(aiData[0]);

  printf("Number of element = %d",iTotalElement);

  return 0;
}
```

Question 63: Write a c Program to Check Whether a Number is Prime or Not?

A prime number is a positive natural number, whose value greater than 1 and has only two factors 1 and the number itself.

Algorithm to check prime number using division method

Step 1 → Take the number n
Step 2 → Divide the number n with (2, n-1) or (2, n/2) or (2, sqrt(n)).
Step 3 → if the number n is divisible by any number between (2, n-1) or (2, n/2) or (2, sqrt(n)) then it is not prime
Step 4 → If it is not divisible by any number between (2, n-1) or (2, n/2) or (2, sqrt(n)) then it is a prime number.

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

#define PRIME_NUMBER  1

int IsPrimeNumber(int iNumber)
{
   int iLoop = 0;
   int iPrimeFlag = 1;
   int iLimit = sqrt(iNumber); // calculate of square root n

   if(iNumber <= 1)
   {
      iPrimeFlag = 0;
   }
   else
   {
     for(iLoop = 2; iLoop <= iLimit; iLoop++)
        {
          if((iNumber % iLoop) == 0) // Check prime number
            {
              iPrimeFlag = 0;
              break;
            }
        }
    }
    return iPrimeFlag;
}

int main(int argc, char *argv[])
{
   int iRetValue = 0;
   int iNumber = 0;

   printf("Enter the number : ");
   scanf("%d",&iNumber);

   iRetValue = IsPrimeNumber(iNumber);

   if (iRetValue == PRIME_NUMBER)
      printf("\n\n%d is prime number..\n\n", iNumber);
   else
      printf("\n\n%d is not a prime number..\n\n", iNumber);
   return 0;
}
```

Question 64: How to find the size of the structure in c without using sizeof operator?

Method 1:

When incrementing the pointer then pointer increase a block of memory (block of memory depends on pointer data type). So here we will use this technique to calculate the sizeof structure.

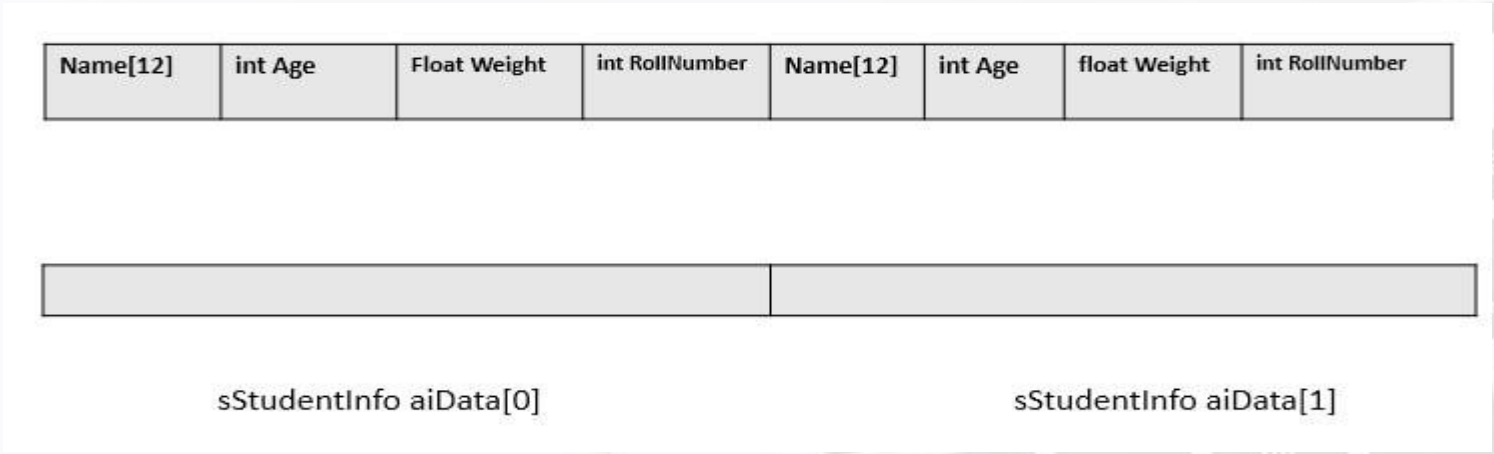
1. First, create the structure.
2. Create a pointer to structure and assign the NULL pointer.
3. Increment the pointer to 1.

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


typedef struct {

    char Name[12];
    int Age;
    float Weight;
    int RollNumber;

}sStudentInfo;



int main(int argc, char *argv[])
{

  //Create pointer to the structure
   sStudentInfo *psInfo  = NULL;

  //Increment the pointer
   psInfo++;

   printf("Size of structure  =  %u\n\n",psInfo);

}
```

Method 2:

We can also calculate the size of the structure using the pointer subtraction. In the previous article “all about the pointer“, we have read that using the pointer subtraction we can calculate the number of bytes between the two pointers.

1. First, create the structure.
2. Create an array of structure, Here aiData[2].
3. Create pointers to the structure and assign the address of the first and second element of the array.
4. Subtract the pointers to get the size of the structure.




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


typedef struct {

   char Name[12];
   int Age;
   float Weight;
   int RollNumber;

}sStudentInfo;

int main(int argc, char *argv[])
{
  //create an array of structure;
   sStudentInfo aiData[2] = {0};


  //Create two pointer to the integer
  sStudentInfo *piData1 = NULL;
  sStudentInfo *piData2 = NULL;
  int iSizeofStructure = 0;

  //Assign the address of array first element to the pointer
  piData1 = &aiData[0];

  //Assign the address of array third element to the pointer
  piData2 = &aiData[1];

 // Subtract the pointer
  iSizeofStructure = (char*)piData2 - (char *)piData1;

  printf("Size of structure  =  %d\n\n",iSizeofStructure);
  return 0;

}
```

Question 65: What is meant by structure padding?

In the case of structure or union, the compiler inserts some extra bytes between the members of structure or union for the alignment, these extra unused bytes are called padding bytes and this technique is called padding.

Padding has increased the performance of the processor at the penalty of memory. In structure or union data members aligned as per the size of the highest bytes member to prevent from the penalty of performance.

Note: *Alignment of data types mandated by the processor architecture, not by language.*

Question 66: How to pass a 2d array as a parameter in C?

In C language, there are a lot of ways to pass the 2d array as a parameter. In below section, I am describing few ways to pass the 2d array as a parameter to the function.

Passing 2d array to function in c using pointers

The first element of the multi-dimensional array is another array, so here, when we will pass a 2D array then it would be split into a pointer to the array.

For example,

If int aiData[3][3], is a 2D array of integers, it would be split into a pointer to the array of 3 integers (int (*)[3]).

```
#include <stdio.h>

#define ARRAY_ROW      3
#define ARRAY_COL      3

void ReadArray(int(*piData)[ARRAY_COL])
{
     int iRow = 0;
     int iCol = 0;

     for (iRow = 0; iRow < ARRAY_ROW; ++iRow)
       {
         for (iCol = 0; iCol < ARRAY_COL; ++iCol)
           {
             printf("%d\n", piData[iRow][iCol]);
           }
       }
}

int main(int argc, char *argv[])
{
   int aiData[ARRAY_ROW][ARRAY_COL] = { { 1, 2, 3 }, { 4, 5, 6 }, { 7, 8, 9 } };

   ReadArray(aiData); //Pass array as a parameter

 return 0;
}
```

Passing 2d array to function with row and column

In which prototype of the function should be same as the passing array. In another word, we can say that if int aiData[3][3] is a 2D array, the function prototype should be similar to the 2D array.

```
#include <stdio.h>

#define ARRAY_ROW      3
#define ARRAY_COL      3

void ReadArray(int aiData[ARRAY_ROW][ARRAY_COL])
{
   int iRow = 0;
   int iCol = 0;

   for (iRow = 0; iRow < ARRAY_ROW; ++iRow)
      {
        for (iCol = 0; iCol < ARRAY_COL; ++iCol)
         {
            printf("%d\n", aiData[iRow][iCol]);

          }
         }
}

int main(int argc, char *argv[])
{
   //Create an 2D array
   int aiData[ARRAY_ROW][ARRAY_COL] = { { 1, 2, 3 }, { 4, 5, 6 }, { 7, 8, 9 } };

    //Pass array as a parameter
    ReadArray(aiData);
    return 0;
}
```

Passing 2d array to function, using the pointer to 2D array

If int aiData[3][3] is a 2D array of integers, then &aiData would be pointer the 2d array that has 3 row and 3 column.

```
#include <stdio.h>

//Size of the created array
#define ARRAY_ROW      3
#define ARRAY_COL      3


void ReadArray(int(*piData)[ARRAY_ROW][ARRAY_COL])
{
   int iRow = 0;
   int iCol = 0;

   for (iRow = 0; iRow < ARRAY_ROW; ++iRow)
      {
        for (iCol = 0; iCol < ARRAY_COL; ++iCol)
            {
              printf("%d\n", (*piData)[iRow][iCol]);

            }
       }

}


int main(int argc, char *argv[])
{
   //Create an 2D array
   int aiData[ARRAY_ROW][ARRAY_COL] = { { 1, 2, 3 }, { 4, 5, 6 }, { 7, 8, 9 } };

  //Pass array as a parameter
  ReadArray(&aiData);

  return 0;
}
```

Question 67: What will be output of below code?

```
#include<stdio.h>
int main()
{
static int data = 5;
printf("%d ",data--);
if(data)
main();

return 0;
}
```

Answer:
5 4 3 2 1

static variable initialized once, so it is retained the value between the function calls.

Question 68: Write a C program without using semicolon to print 'Hello world'

Using the conditional statements if/switch/while, we can print the statement. Let's see the below example,

Using the if

```
#include <stdio.h>

int main(int argc, char *argv[])
{
    if(printf("Hello world"))
   { }
}
```

Using the while

```
#include <stdio.h>

int main(int argc, char *argv[])
{
   while(!printf("Hello world "))
   {
   }
}
```

Using the switch

```
#include <stdio.h>

int main(int argc, char *argv[])
{
   switch(printf("Hello world"))
   {
   }
}
```

Question 69: What are differences between sizeof operator and strlen function?

A sizeof is a C operator that can calculate the size of a string constant including the null character. A strlen is library function which can only calculate the size of a string constant, it excludes the null character.

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


int main(int argc, char *argv[])
{
    int len = 0;

    len=strlen("aticleworld");

    printf(" Using strlen %d\n",len);

    len=sizeof("aticleworld");

    printf(" Using sizeof %d\n",len);

    return 0;
}
```

Question 70: What is an lvalue?

An lvalue is an expression to which a value can be assigned. We can write lvalue expression either left side or right side of the assignment (=) statement.

Let's see an example,

int iData1 = 10; (compiler interprets iData1 as lvalue)

int iData2 = iData1; (compiler interprets iData2 as lvalue and iData1 is rvalue)

10 = iData1; (compiler interprets iData1 as rvalue but 10 as an lvalue. Here you will get a compiler error because lvalue should have storage)

Question 71: What are preprocessor directives?

Preprocessor directive starts with the # symbol and there is no need to put the semicolon (;) at the end preprocessor directive. In C language, preprocessor directive has a lot of application like in header file inclusion or constant declaration .etc.

We can also use preprocessor directive as file guard. Whenever you compile the code, the preprocessor examines the code before actual compilation and resolves all these directives.

Example,

```
//Preprocessor as a constant

#define COLOUR   10


// Preprocessor include stdio.h

#include<stdio.h>


// Preprocessor is used below as file guard

#ifndef MAIN_H__
#define MAIN_H__

//Header file content

#endif // MAIN_H__
```

Question 72: How can you avoid including a header more than once?

Using the file guard, we can protect a header file for multiple inclusion. In C using the preprocessor directive, we can achieve this functionality.

For example,

```
#ifndef _HEADER_
#define _HEADER_

void add();
void sub();
void mul();

#endif
```

Question 73: What is the difference between array_name and &array_name?

To understand this question let's take an example, suppose arr is an integer array of 5 elements.int arr[5];

If you print arr and &arr then you found the same result but both have different types.

arr => The name of the array is a pointer to its first element. So here arr split as the pointer to the integer.

&arr => It split into the pointer to an array that means &arr will be similar to int (*)[5];

```
#include<stdio.h>

int main()
{
  int arr[5];

  printf("arr =  %u\n\n", arr);

  printf("&arr = %u\n\n", &arr);

  printf("arr+1 =  %u\n\n", arr+1);

  printf("&arr+1 = %u\n\n", &arr+1);


  return 0;
}
```

When you compile the above code, you will find arr and &arr is same but the output of arr+1 and &arr+1 will be not same due to the different pointer type.

Question 74: Can you subtract pointers from each other? Why would you?

In C language, we can subtract one pointer from another pointer. This technique is useful when we need to calculate the number of bytes between the two pointers. But we subtract two pointers only when they pointing the same array or memory block. The result of this subtraction gives the number of elements that separating them.

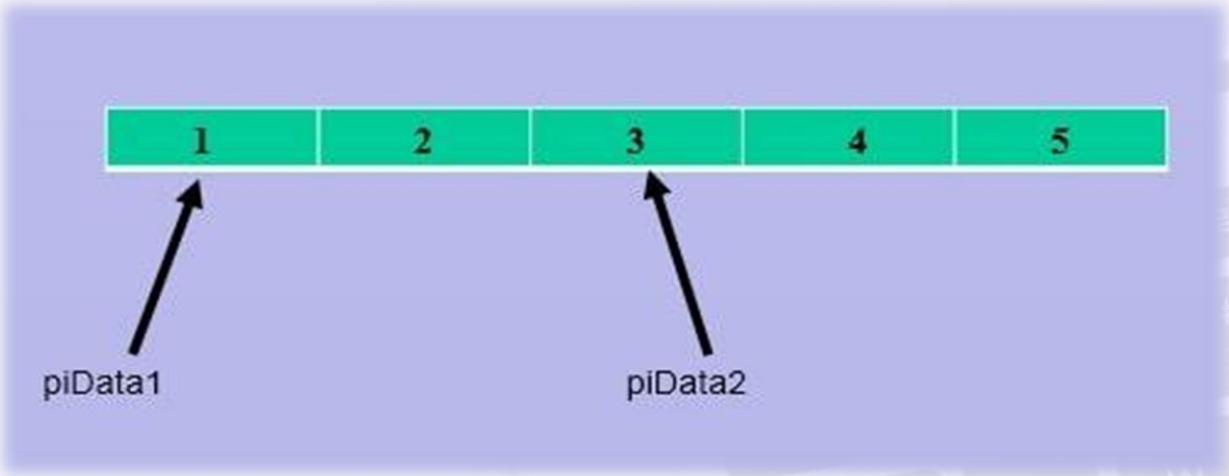
Note: There are is no meaning to the addition of two pointers.

Let's consider two example which describes the subtraction of two number.

Example 1:In this example, we will calculate the size of the structure to subtract the pointers.

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


typedef struct {

    int Age;
    float Weight;

}sInfo;



int main(int argc, char *argv[])
{
  //create an array of structure;
  sInfo JhonFamilyInfo[2];

  //Create pointer to the structure
  sInfo *psInfo  = NULL;

   int iSizeofStructure = 0;

    //Assign the address of array to the pointer
    psInfo = JhonFamilyInfo;

    // Subtract the pointer
    iSizeofStructure = (char*)(psInfo + 1) - (char*)(psInfo);

    printf("Size of structure  =  %d\n\n",iSizeofStructure);

}
```

Example 2:
In this example, we will calculate the number of elements that separating the pointers or you can say calculate the pointer offset.




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

int main(int argc, char *argv[])
{
   //create an array of structure;
    int aiData[] = {1,2,3,4,5};


   //Create two pointer to the integer
     int *piData1 = NULL;
     int *piData2 = NULL;

     int iOffset = 0;

     //Assign the address of array first element to the pointer
      piData1 = &aiData[0];

      //Assign the address of array third element to the pointer
      piData2 = &aiData[2];

      // Subtract the pointer
      iOffset = piData2 - piData1;

     printf("pointer offset  =  %d\n\n",iOffset);

     return 0;

}
```

Question 75: Can you add pointers together?

As like the pointer subtraction there is no meaning of pointer addition. Pointers contains the addresses. Adding two addresses makes no sense because you have no idea what you would point to.

Question 76: What happens if you free a pointer twice?

A free function is used to deallocate the allocated memory. If piData (arguments of free) is pointing to a memory that has been deallocated (using the free or realloc function), the behavior of free function would be undefined.

Freeing the memory twice is more dangerous then memory leak, so it is very good habits to assigned the NULL to the deallocated pointer because the free function does not perform anything with the null pointer.

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

int main()
{
int *piData = NULL;

piData = malloc(sizeof(int) * 10); //creating integer of size 10.

free(piData); //free the allocated memory

free(piData); //free the allocated memory twice

return 0;

}
```

Question 77: Can math operations be performed on a void pointer?

According to c standard arithmetic operation on void pointers is illegal that means the C standard doesn't allow pointer arithmetic with void pointers. However, In GNU C, addition and subtraction operations are supported on void pointers to assuming the size of the void is 1.

```
#include<stdio.h>
int main()
{
   int aiData[3] = {100, 200 ,300};

   void *pvData = &aiData[1]; //address of 200

   pvData += sizeof(int);

   printf("%d", *(int *)pvData);

   return 0;
}
```

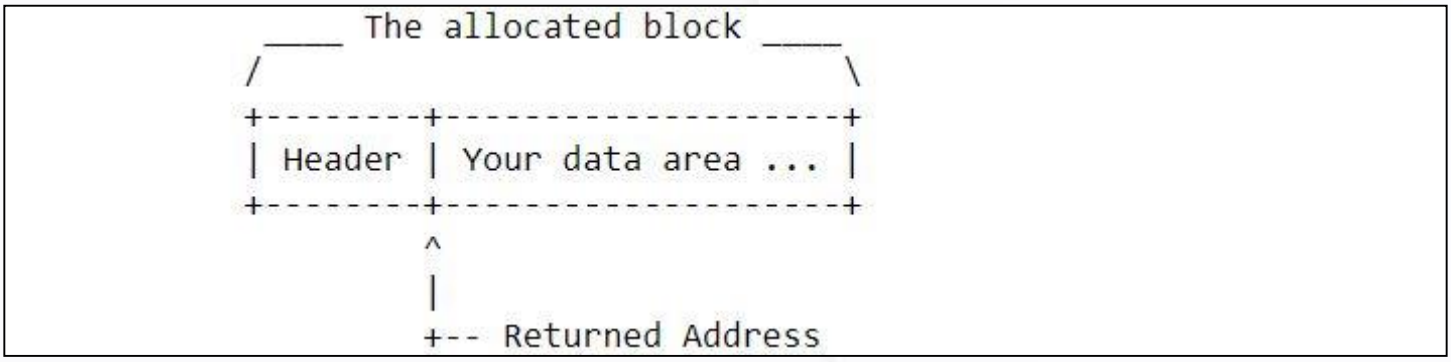
Output: 300 or compiler error.

Explanation: When we compile the code then some compiler throw the compiler error but some compiler compiled the code and print 300 as output to assume the size of the void 1.

Note: Don't perform the arithmetic operation on the void pointer. As per the C standard sizeof is not applicable on void but in GNU C we can calculate the size of the void and sizeof operator return 1.

Question 78: How does free() know how much memory to release?

When we call the memory management functions (malloc, calloc or realloc) then these functions keep extra bytes for bookkeeping. Whenever we call the free function and pass the pointer that is pointing to allocated memory, the free function gets the bookkeeping information and release the allocated memory. Anyhow if you or your program change the value of the pointer that is pointing to the allocated address, the calling of free function give the undefined result.



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

int main()
{
  char *pcBuffer = NULL;

   pcBuffer  =  malloc(sizeof(char) *  16); //Allocate the memory

   pcBuffer++; //Increment the pointer

  free(pcBuffer); //Call free function to release the allocated memory

  return 0;
}
```

Question 79: Write a self-producing code in C?

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

int main(int argc, char *argv[])
{
  FILE * pFile;
  int c;
  pFile=fopen (__FILE__,"r");
  if (pFile==NULL) perror ("Error opening file");
  else
  {
    do {
      c = getc (pFile);
      printf("%c",c);
    } while (c != EOF);
    fclose (pFile);
  }
  return 0;

}
```

Question 80: Rotate bits of a number in C?

Like the assembly in C language there is no operator to rotate the bits, so if we require rotating a bit, then we have to do it manually.

Basically, bit rotation is similar to the shift operation except that in shift operation the bits that fall off at one end are put back to the other end.

There is two type of rotation possible left and right. In the left rotation, the bits that fall off at left end are put back at right end and in right rotation, the bits that fall off at right end are put back at the left end.

Example:

If data is stored using 8 bits, then left the rotation of a data 32(00100000) by 2 becomes 128 (10000000). As similar to left rotation, if data is stored using 8 bits, then right rotation of the data 32(00100000) by 2 becomes 8 (00001000).

```
#include <stdio.h>

#define INT_BITS 32

#define ROTATE_LEFT(pos, data) ((data << pos)|(data >> (INT_BITS - pos)))

#define ROTATE_RIGHT(pos, data) ((data >> pos)|(data << (INT_BITS - pos)))


int main()
{
  int pos = 2; // Number of rotation

  int data = 32; //data which will be rotate

  printf("%d Rotate Left by %d is ", data, pos);
  printf("%d \n", ROTATE_LEFT(pos, data));

  printf("%d Rotate Right by %d is ",data, pos);
  printf("%d \n", ROTATE_RIGHT(pos, data));

  return 0;
}
```

Question 81: Compute the minimum (min) or maximum (max) of two integers without branching?

We can find the minimum (min) or maximum (max) number without the branching with the help of a bitwise operator.

Let's assume “a” and “b” are integers numbers and “result” is another integer variable that contains the result of the computation.

So to compute the minimum number we have to write the below expression.

```
result = b ^ ((a ^ b) & -(a < b)); // min(a, b)
```

In above expression,if a < b, then -(a < b) become -1, so it behave like below expression

```
result = b ^ ((a ^ b) & ~0);

result =  b ^ a ^ b; // b^b is zero

result = a ^ 0; // oring with 0 does not effect

result = a; //minimum number
```

Compute the maximum number we have to write the below expression.

```
result = a ^ ((a ^ b) & -(a < b)); // max(a, b)
```

In above expression, if a > b, then -(a > b) become 0, so it behave like below expression

```
result = a ^ ((a ^ b) & -(0));

result = a ^ 0; // oring with 0 does not effect

result = a; //Maximum number
```

Question 82: Multiply a number by 2 using bitwise operation.

Left shifting of a data (number) by 1 is equivalent to data*2. In data, every bit is a power of 2, with each shift we are increasing the value of each bit by a factor of 2.

```c
#include <stdio.h>


int main()
{

  unsigned int data = 15; //value of data

  data = data << 1; // equivalent to data * 2

  printf("data = %d\n", data);

  return 0;
}
```

Question 83: Divide a number by 2 using bitwise operation.

Right shifting of a data (number) by 1 is equivalent to data/2. In data, every bit is a power of 2, with each right shift we are reducing the value of each bit by a factor of 2.

```c
#include <stdio.h>

int main()
{

  unsigned int data = 16; //value of data

  data = data >> 1; // equivalent to data/2

  printf("data = %d\n", data);

  return 0;
}
```

Question 83: Multiply a given Integer with 3.5 using bitwise operation.

We know that multiplication is a type of addition, so we can multiply a given integer (data) with 3.5 using the following operation, (2 *data) + data + (data/2).

```c
#include <stdio.h>

int main()
{
  unsigned int data = 10; //value of data

  data = (data<<1) + data + (data>>1);; // equivalent to data * 3.5

  printf("data = %d\n", data);

  return 0;
}
```

Question 84: Clear all bits from MSB to ith bit.

Here I have supposed data is stored using 8 bits.

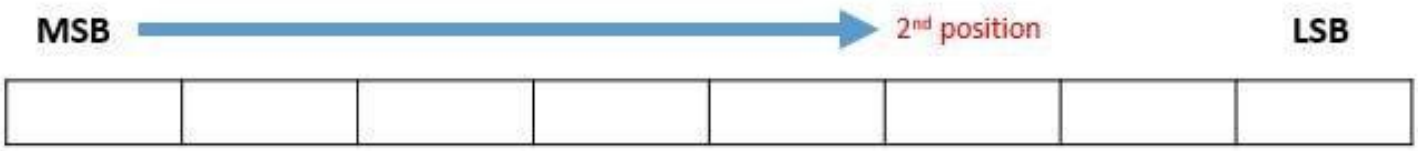
let's assume the ith position is 2.

mask =(1 <<(i+1)); // give you 00001000

so now if we subtract 1 from the mask (mask = mask – 1), then we will get 00000111

Using the mask, now we can clear MSB to ith bits of data (15).

data = data & mask; // Now bits are clear



```c
#include <stdio.h>

int main()
{
  unsigned int mask = 0; // mask flag

  unsigned int i = 2; // ith position till u want to clear the bits

  unsigned int data = 15; //value of data


  mask = (1 << (i+1)); //Shift 1 ith position

  mask = mask -1 ; //give us 00000111

  //Now clear all bits from msb to ith position
  data  = data & mask;

  printf("data = %d\n", data);

  return 0;
}
```

Question 85: Clear all bits from LSB to ith bit.

To clear all bits of a data from LSB to the ith bit, we have to perform AND operation between data and mask (flag) having LSB to ith bit 0.

To create a mask, first left shift 1 (i+1) times.

mask =(1 << (i+1)); // give you 00001000

Now if we minus 1 from that, all the bits from 0 to i become 1 and remaining bits become 0.

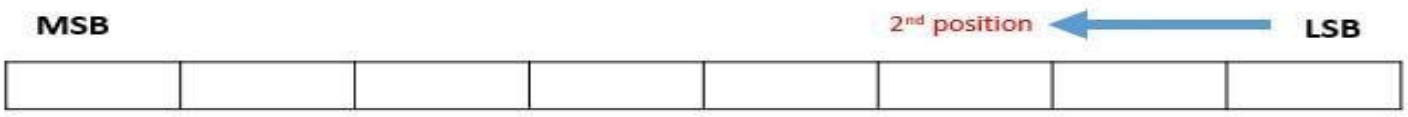
mask = mask – 1 ; // give you 00000111

After that perform complement operation on the mask, all the bits from 0 to i become 0 and remaining bits become 1.

mask = ~mask; //give you 11111000

Now just simply perform anding operation between mask and data to get the desired result.

data = data & mask; // Now bits are clear from LSB to ith position



```
#include <stdio.h>


int main()
{
  unsigned int mask = 0; // mask flag

  unsigned int i = 2; // ith position till u want to clear the bits

  unsigned int data = 15; //value of data


  mask = (1 << (i+1)); //Shift 1 ith position

  mask = mask -1 ; //give us 00000111

   mask = ~mask; //give us 11111000

  //Now clear all bits from msb to ith position
  data  = data & mask;

  printf("data = %d\n", data);

  return 0;
}
```

Question 86: Swap two nibbles of a byte.

A nibble consists four bits, sometime interviewer asked the question to swap the nibble of a byte. It is a very easy question, here << (left shift) and >> (right shift) operators are used to swap the nibble.

```
#include <stdio.h>

//Macro to swap nibbles

#define SWAP_NIBBLES(data) ((data & 0x0F)<<4 | (data & 0xF0)>>4)

int main()
{
    unsigned char value = 0x23; //value in hex

    printf("0x%x", SWAP_NIBBLES(value)); //print after swapping

    return 0;
}
```

Question 87: Write a self-producing code in C?

You just need to open the source file in reading mode. If the source file opens successfully then read the file data byte by byte and display it on the console screen.

```
#include <stdio.h>


int main(int argc, char *argv[])
{
  FILE * pFile; //File pointer

  int readData = 0;//file data

  pFile=fopen (__FILE__,"r"); //open source file in read mode

  if (pFile==NULL)
  {
            perror("Error opening file");
  }
  else
  {
    do {

      readData = getc(pFile); //read file data

      printf("%c",readData); //print data on console

    } while (readData != EOF);

    fclose (pFile); //close open file
  }


  return 0;

}
```

Question 88: What will be output of below program?

```
#include<stdio.h>

int main()
{
float f = 1.1;
double d = 1.1;

if(f==d)
{
  printf("Welcome");
}
else
{
  printf("Bye");
}

return 0;
}
```

Explanation: The precision of float less than double, you can say float stores 1.1 with less precision than double.

Question 89: What will be output of below program?

```
#include<stdio.h>

int main()
{
extern int data;

data = 1993;

printf("%d",data);

return 0;
}
```

Output:
undefined reference to `data'

Explanation:
In above example, we have declared an integer variable using extern. At the time of linking linker search the address of the variable but here we have only declared the variable that is why getting a linker error.

Question 90: What will be output of below program?

```
#include<stdio.h>

int main()
{

char *ch;

printf("%d %d ",sizeof(*ch),sizeof(ch));

return 0;
}
```

Explanation:

The sizeof() operator gives the number of bytes taken by its operand. ch is a pointer to a character, which needs one byte for storing its value (a character). Hence sizeof(*ch) gives a value of 1. Since ch is a pointer to a character, so sizeof(ch) gives the size of the pointer that depends on your development environment and system.

Question 91: What will be output of below program?

```
#include<stdio.h>

int main()
{
unsigned int i=10;

while(i-->=0)
printf("%u ",i);

return 0;
}
```

Explanation:

Since i is an unsigned integer. So expression i-- >=0 will always be true and while loop behaves like the infinite loop.

Question 92: What will be output of below program?

```
#include<stdio.h>

int main()
{

int data = 10;

printf("%d",++data++);

return 0;
}
```

Output:

Compiler error:

Lvalue required in function main

Explanation:

++data yields an rvalue. For postfix ++ to operate an lvalue is required.

Question 93: What will be output of below program?

```
#include<stdio.h>

int main()
{

register int data =2;

printf("Address of a = %d",&data);

printf("Value of a = %d",data);

return 0;
}
```

Output:
Compiler Error

Explanation:
&(address of) operator cannot be applied on the register variables.

Question 94: Difference between structure and union?

The key difference between structure and union is that structure allocate enough space to store all the fields but unions only allocate enough space to store the largest field. In union, all fields are stored in the same space.

In below table, I have listed some common difference between structure and union.

union	structure
union keyword is used to define the union type.	structure keyword is used to define the union type.
Memory is allocated as per largest member.	Memory is allocated for each member.
All field share the same memory allocation.	Each member have the own independent memory.

We can access only one field at a time.
 	
union Data {
int a; // can't use both a and b at once
char b;
} Data;
 	
union Data x;
x.a = 3; // OK
x.b = 'c'; // NO! this affects the value of x.a
	We can any member any time.
	
	struct Data {
	int a; // can use both a and b simultaneously
	char b;
	} Data;
	
	struct Data y;
	y.a = 3; // OK
	y.b = 'c'; // OK

Altering the value of the member affect the value of the other member.	Altering the value of the member does not affect the value of the other member.
Flexible array is not supported by the union.	Flexible array is supported by the structure.
With the help of union we can check the endianess of the system.	We cannot check the endianess. https://aticleworld.com/

Question 95: What will be output of below program?

```
#include<stdio.h>
 int main()
 {
    char *p ="Aticleworld";

    printf("%c\n",*(&(*(p+1))+1)+1);

   return 0;
 }
```

Question 96: What will be output of below program?

```
#include<stdio.h>

 #define MAX(x,y) (((x) > (y)) ? (x) : (y))

 #define FINDMAX(a,b,c) MAX(a,MAX(b,c))

 int main()
 {
   printf("%d", FINDMAX(12,345,232));

   return 0;
 }
```

Question 97: What will be output of below program?

```
#include<stdio.h>

 int main()
 {
   while(1)
   {
     if(!printf("%d",printf("%d",109)))
        break;
      else
        printf("ok");
      break;
    }

  return 0;
 }
```

Question 98: What will be output of below program?

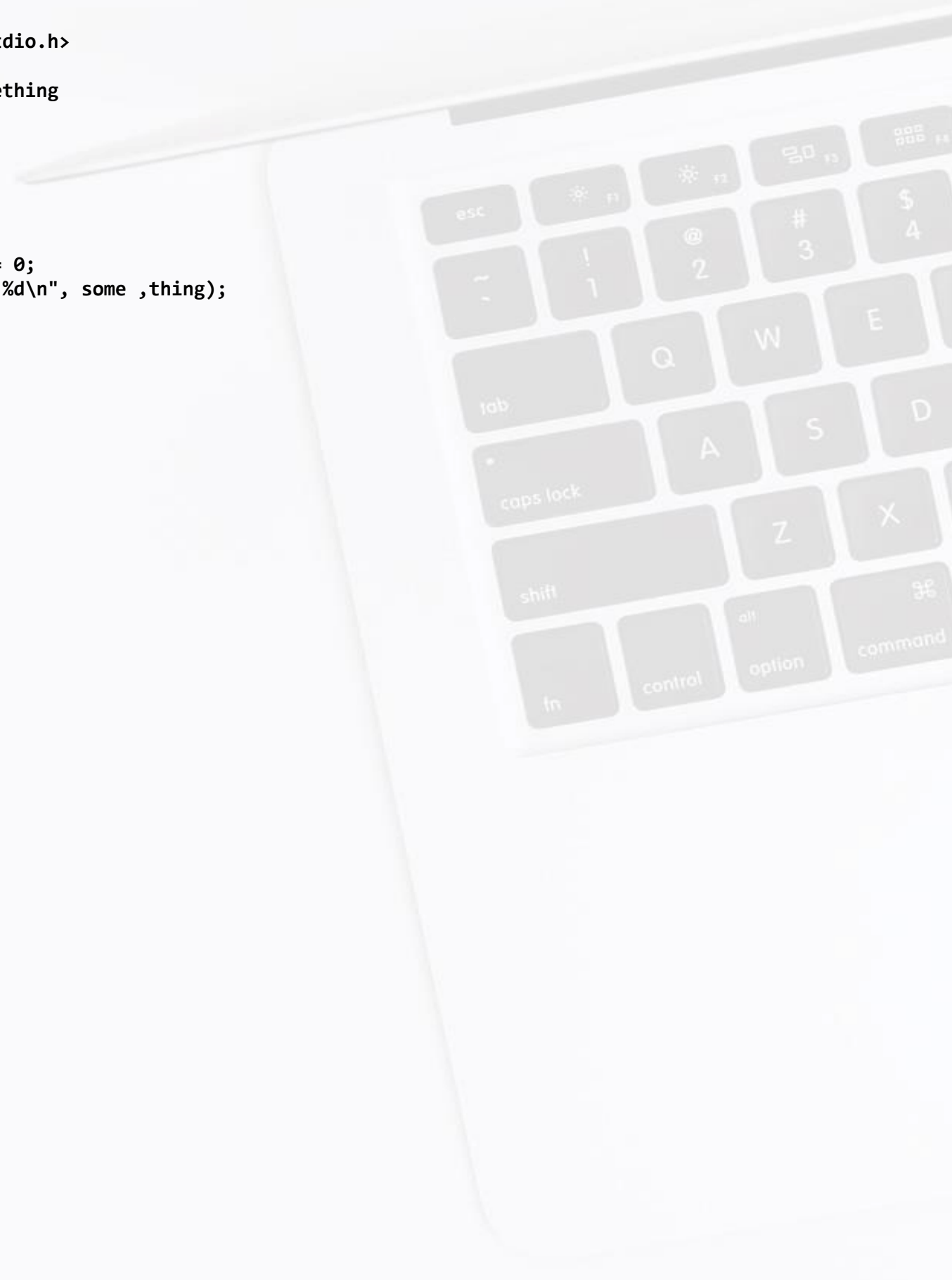
```
#include<stdio.h>

int main()
{

int arr[]={1,6,0,4,5,6},i;

int *ptr = arr;

*((char*)ptr)++;

*ptr=1;

printf("%d %d  ",arr[0],arr[1]);

return 0;
}
```

Question 99: What will be output of below program?

```
#include<stdio.h>

int Inc(int j)
{
  ++j;
  return j;
}


int main()
{
 int data;

 data = 3;

 printf("%c",63+Inc(Inc(printf("%d",data))));


return 0;
}
```

Question 100: What will be output of below program?

```
#include<stdio.h>

#ifdef something
int some=0;
#endif

int main()
{

int thing = 0;
printf("%d %d\n", some ,thing);

return 0;
}
```



About the Author:

I am an embedded c software engineer and a corporate trainer, currently working as senior software engineer in one of the largest Software consulting company. I have experience of working on different microcontrollers viz. STM32, LPC, PIC, AVR and 8051, drivers (USB and virtual com-port), POS device (VeriFone), payment gateway (global and first data),cinema4d plugin.

Amlendra Kumar