100+ C interview questions, your interviewer might ask

You are looking for C interview questions or tricky C interview questions, then you are at the right place. Here I have tried to create a collection of good C Interview questions. I have spent many hours to create these C interview questions. So I hope you will enjoy these tricky C interview questions and you will learn new concepts of programming in C interview questions. All the best for your C interview.

Q) What is the difference between declaration and definition of a variable?

Ans:

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 times 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 other words, 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.

Q) What is the difference between global and static global variables?

Ans:

Global and static global variables have different linkages. It is the reason global variables can be accessed outside of the file but the static global variable only accesses within the file in which it is declared.

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

For more details, you can see below-mentioned articles,

- **Internal, External and None linkage in c.**
- **Local, Static and Global variables in C.**

Q) What are storage classes in C language?

Ans:

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 the variable's value is stored in the form of bits. The storage classes decide where these variable values will store like in CPU register, stack memory, BSS or DS.

There are four storage classes available in C programming.

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

For more details, you can see below-mentioned articles,

- **Storage classes in C.**
- **Memory Layout of C program.**

Q) Differentiate between an internal static and external static variable?

Ans:

In C language, the external static variable has the internal linkage and the internal static variable has no linkage. It is the reason they have a different scope but both will alive throughout the program.

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

Q) What is the difference between typedef & Macros?

Ans:

Let's see the short description of the **typedef** and **macro** to understand the difference between them.

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 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. So in the below example, Mydata is unsigned int variable.

```
UnsignedInt Mydata;
```

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 problems with the macro is that there is no type checking. Generally, the macro is used to create the alias, in C language. A macro is also used as a file guard in C and C++.

Syntax,

```
#define MACRO_NAME MACRO_VALUE
```

eg,

```
#define VALUE 10
```

Now VALUE becomes 10 in your program. You can use the VALUE in place of the 10.

For more details, you can see below-mentioned articles,

- **Macro in C.**
- **typedef vs #define in C.**
- **typedef in C.**

Q) What is the output of the below C code?

```
#include<stdio.h>
int main()
{
typedef auto int myAutoInt;
myAutoInt data = 4;
printf("%d",data);
return 0;
}
```

Ans:

compiler error.

Explanation:

typedef already consider as partial storage class, so you can not mix two storage classes at a time.

Q) What is the variable in C?

Ans:

A variable in C defines a location name where you can put value and you can use these values whenever required in the program. In other words, you can say that variable is a name (or identifier) which indicates some physical address in the memory, where data store in the form of bits of the 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.

Q) 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.

Ans:

```
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
```

Q) What are the data types in C?

Ans:

A data type is a classification of data that tells the compiler or interpreter how the programmer intends to use the data. In other words, you can say that it defines the size (BYTE) and the range of a **variable**.

Classification of the data types in C language

1. Pre-define data types (int, char, float, etc)
2. User-define data types (struct, union, enum)

In C language, different data types have different ranges. The range varies from compiler to compiler. In the 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

For more details, you can see below-mentioned articles,

- **Data types in C**
- **Format specifiers in C.**
- **Elements of C language.**

Q) 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 *));

Q) What are the uses of the keyword static?

Ans:

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.

1. A static variable only initializes once, so a variable declared static within the body of a function maintains its prior value between function invocations.

2. A global variable with static keyword has 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.

3. By default in C language, the 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.

Q) What are the different types of linkage?

Ans:

The C language has three types of linkage, **External Linkage, Internal Linkage**, and **None Linkage.**

Q) Can static variables be declared in a header file?

Ans:

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

Q) Size of the integer depends on what?

Ans:

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 the size of integer depends on the type of the compiler which has written by the 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 to use the C99 integer data types (uin8_t, uin16_t, uin32_t ..) in place of standard int.

Q) Are integers signed or unsigned?

Ans:

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

For more details on signed and unsigned integer, check out:
A closer look at signed and unsigned integers in C

Q) What is a difference between unsigned int and signed int in C?

Ans:

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 the 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 the 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 shows the undefined behavior but for the unsigned type, it would be Data x 2pos.

Q) What is the difference between a macro and a function?

Ans:

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.

Q) What is the output of the below C code?

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

```
#define PRINT(var,data) do {\
if(var < data)\
{\
printf("Aticleworld");\
++var;\
}\
}while(1);
int main()
{
PRINT(0,2);
return 0;
}
```

Output: compiler Error.

Explanation: When macro will be expanded, var will be replaced with 0 (Rvalue). Because you are trying to increment the Rvalue, you will get a compiler error. You can read this article for more detail, **post, and pre-increment operators.**

Q) What do you mean by enumeration in C?

Ans:

An **enum** in C is a user-defined data type. It consists set of named constant integers. Using the enum keyword, we can declare an enumeration type by using the enumeration tag (optional) and a list of named integer.

Basically, we used the enum to increase the code readability and with enum easy to debug the code as compared to symbolic constant (macro). The most important property of enum is that it follows the scope rule and the compiler automatically assigns 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};
```

For more details, you can see below-mentioned articles,

- **Macro in C.**
- **enum in C.**
- **typedef vs #define in C.**

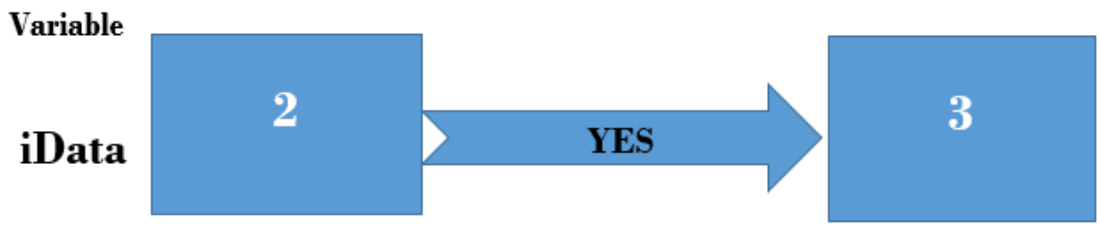
Q) What does the keyword const mean?

Ans:

The const qualifier only gives the direction to the compiler that the value of qualify object cannot be changed. In simple words, const means not modifiable (cannot assign any value to the object at the run time).

Syntax:

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



For more details, you can see below-mentioned articles,

- **const in C.**
- **Difference between const and volatile.**
- **Question-Related to const.**

Q) When should we use const in a C program?

Ans:

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

- In the call by reference function argument, if you don't want to change the actual value which has passed in function.

  ```
  int PrintData ( const char *pcMessage);
  ```

- In some places, const is better than macro because const handled by the compiler and have a type checking.

  ```
  const int ciData = 100;
  ```

- In the case of the I/O and memory-mapped register, const is used with the volatile qualifier for efficient access.

  ```
  const volatile uint32_t *DEVICE_STATUS = (uint32_t *) 0x80102040;
  ```

- When you don't want to change the value of an initialized variable.

Q) What is the meaning of the below declarations?

```
1. const int a;
2. int const a;
3. const int *a;
4. int * const a;
5. int const * a const;
```

Ans:

1. The "a" is a constant integer.
2. Similar to first, "a" is a constant integer.
3. Here "a" is a pointer to a const integer, the value of the integer is not modifiable, but the pointer is not modifiable.
4. Here "a" is a const pointer to an integer, the value of the pointed integer is modifiable, but the pointer is not modifiable.
5. Here "a" is a const pointer to a const integer that means the value of the pointed integer and pointer both are not modifiable.

Q) What is the output of the below C program?

```
#include <stdio.h>
int main()
{
int pos = 14;
float data = 1.2;
printf("%*f",pos,data);
return 0;
}
```

Ans:

The output of the above code will be 1.200000 with 6 space.

Explanation:

Here 1.200000 is printing with, 6 spaces, because by giving * in printf we can specify an additional width parameter, here 'pos' is the width and 'data' is the value. if the number is smaller than the width then rest is filled with spaces.

Q) Differentiate between a constant pointer and pointer to a constant?

Ans:

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;
```

eg,

```
int *const ptr; //constant pointer to integer
```

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

```
#include<stdio.h>
int main(void)
{
int var1 = 10, var2 = 20;
//Initialize the pointer
int *const ptr = &var1;
//Try to modify the pointer value
ptr = &var2;
printf("%d\n", *ptr);
return 0;
}
```

Pointer to a constant:

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

A constant pointer is declared as follows :

```
Data_Type const* Pointer_Name;
eg,
int const *ptr// pointer to const integer
```

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

```
#include<stdio.h>
int main(void)
{
int var1 = 100;
// pointer to constant integer
```

```
const int* ptr = &var1;
//try to modify the value of pointed address
*ptr = 10;
printf("%d\n", *ptr);
return 0;
}
```

Q) What are the post-increment and decrement operators?

Answer:

When we use a post-increment (++) operator on an operand then the result is the value of the operand and after getting the result, the value of the operand is incremented by 1. The working of the post-decrement (-) operator is similar to the **post-increment operator** but the difference is that the value of the operand is decremented by 1.

***Note**: incrementation and decrementation by 1 are the types specified.*

Q) Which one is better: Pre-increment or Post increment?

Answer:

Nowadays compiler is enough smart, they optimize the code as per the requirements. The post and pre-increment both have their 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 the 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 the 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 the pre-increment operator? The answer to this question is that 'A' will be skipped and B will be printed.

```
#include <stdio.h>
int main(void)
{
char acData[5] = {'A','B','C','D','E'};
char *pcData = NULL;
pcData = acData;
printf("%c ",*++pcData);
return 0;
}
```

Output: B

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.

```
#include <stdio.h>
int main(void)
{
char acData[5] = {'A','B','C','D','E'};
char *pcData = NULL;
pcData = acData;
printf("%c ",*pcData++);
return 0;
}
```

Output: A

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 the 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. iLoop = 0.

```
while (a[iLoop ++] != 0)
{
// Body statements
}
```

*Q) Are the expressions *ptr++ and ++*ptr same?*

Ans:

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

```
#include <stdio.h>
int main(void)
{
int aiData[5] = {100,200,30,40,50};
int *ptr = aiData;
*ptr++;
printf("aiData[0] = %d, aiData[1] = %d, *piData = %d", aiData[0], aiData[1], *ptr);
return 0;
}
```

Output: 100, 200, 200

Explanation:

In the above example, two operators are involved and both have different precedence. The precedence of post ++ is higher than the *, so first post ++ will be executed and above expression, *p++ will be equivalent to *(p++). In another word you can say that it is post-increment of address and output is 100, 200, 200.

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

```
int aiData[5] = {100,200,300,400,500};
int *ptr = aiData;
++*ptr;
printf("aiData[0] = %d, aiData[1] = %d, *ptr = %d", aiData[0], aiData[1], *ptr);
return 0;
}
```

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.

*Q) Are the expressions *++ptr and ++*ptr same?*

Ans:

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

```
#include <stdio.h>
int main(void)
{
int aiData[5] = {100,200,30,40,50};
int *piData = aiData;
++*piData;
printf("aiData[0] = %d, aiData[1] = %d, *piData = %d", aiData[0], aiData[1], *piData);
return 0;
}
```

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 other words, we can say it is pre-increment of value and output is 101, 200, 101.

```
#include <stdio.h>
int main(void)
{
int aiData[5] = {100,200,30,40,50};
int *piData = aiData;
*++piData;
printf("aiData[0] = %d, aiData[1] = %d, *piData = %d", aiData[0], aiData[1], *piData);
return 0;
}
```

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 other words, you can say it is pre-increment of address and output is 100, 200,200.

Q) What is the difference between const and macro?

Answer:

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 a 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 the case of call by reference, it prevents modifying the passed object value.

Q) What is a volatile variable in C?

Answer:

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 cache a variable into a register. Reading the value from memory is the only way to check the unpredictable change of the value.

Q) Can we have a volatile pointer?

Ans:

Yes, we can create a volatile pointer in C language.
int * volatile piData; // piData is a volatile pointer to an integer.

Q) The Proper place to use the volatile keyword?

Ans:

Here I am pointing some important places where we need to use the volatile keyword.

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

```
#define COM_STATUS_BIT 0x00000006
uint32_t const volatile * const pStatusReg = (uint32_t*)0x00020000;
unit32_t GetRecvData()
{
//Code to recv data
while (((*pStatusReg) & COM_STATUS_BIT) == 0)
{
// Wait untill flag does not set
}
return RecvData;
}
```

- Sharing the global variables or buffers between the multiple threads.
- Accessing the global variables in an interrupt routine or signal handler.

```
volatile int giFlag = 0;
ISR(void)
{
giFlag = 1;
}
```

```
int main(void)
{
while (!giFlag)
{
//do some work
}
return 0;
}
```

Q) What is the difference between the const and volatile qualifiers in C?

Answer:

The const keyword is compiler-enforced and says that the 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 making any assumption regarding the “a” and in every iteration read the value from the address which is assigned to the variable.

Q) Can a variable be both constant and volatile in C?

Answer:

Yes, we can use both **constant and volatile** together. One of the great use of volatile and const keyword together is at the time of accessing the GPIO registers. In the 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 the 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
```

Q) How to set, clear, toggle and checking a single bit in C?

Ans:

Setting N-th Bit

Setting an N-th bit means that if the N-th bit is 0, then set it to 1 and if it is 1 then leave it unchanged. In C, bitwise OR operator (|) use to set a bit of integral data type. As we know that | (Bitwise OR operator) evaluates a new integral value in which each bit position is 1 only when operand's (integer type) has a 1 in that position.

In simple words, you can say that "Bitwise OR " of two bits is always one if any one of them is one.

That means,

```
0 | 0 = 0
1 | 0 = 1
0 | 1 = 1
1 | 1 = 1
```

Algorithm to set the bits:

```
Number | = (1UL << nth Position);
```

Clearing a Bit

Clearing a bit means that if N-th bit is 1, then clear it to 0 and if it is 0 then leave it unchanged. 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.

That means,

```
0 & 0 = 0
1 & 0 = 0
```

0 & 1 = 0
1 & 1 = 1

Algorithm to clear the bit:

To clear the nth bit, first, you need to invert the string of bits then AND it with the number.

```
Number &= ~(1UL << nth Position);
```

Checking a Bit

To check the nth bit, shift the '1' nth position toward the left and then "AND" it with the number.

An algorithm to check the bit

```
Bit = Number & (1UL << nth)
```


Toggling a Bit

Toggling a bit means that if the N-th bit is 1, then change it to 0 and if it is 0 then change it to 1. 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.

That means,
0 ^ 0 = 0
1 ^ 0 = 1
0 ^ 1 = 1
1 ^ 1 = 0

An algorithm to toggle the bits

```
Number ^= (1UL << nth Position);
```

Q) Detect if two Integers have opposite Signs (Bit Manipulation)

Ans:

Let the given integers are “a” and “b”. The EX-OR of sign bit (MSB) of “a” and “b” will be 1 if the sign bit of “a” is different from the sign bit of “b”. In other words, we can say, EX-OR of “a” and “b” will be negative if “a” and “b” have the opposite signs.

```
bool CheckOppositeSign(int a, int b)
{
bool bRetValue = 0;
bRetValue = ((a ^ b) < 0); // true if a and b have opposite signs
return bRetValue;
}
```

Note: bool exists in the current C – C99.

Q) Write an Efficient C Program to Reverse Bits of a Number?

Ans:

There are a lot of ways to **reverse the bits of a number**, here I am describing three general methods to reverse the bits.

Method 1:

In this method, we will check the set bits of num and run the loop through all the bits of an integer. If we find the ith bits of num is set then just put 1 at the ((INT_BITS – 1) – ith) position of tmp, where INT_BITS is the number of bits of an integer.

```
#define CHAR_BITS 8 // size of character
#define INT_BITS ( sizeof(int) * CHAR_BITS)
//bit reversal function
unsigned int ReverseTheBits(unsigned int num)
{
unsigned int iLoop = 0;
unsigned int tmp = 0; // Assign num to the tmp
int iNumberLopp = (INT_BITS - 1);
for(; iLoop < iNumberLopp; ++iLoop)
{
if((num & (1 << iLoop))) // check set bits of num
{
tmp |= 1 << ((INT_BITS - 1) - iLoop); //putting the set bits of num in tmp
}
}
return tmp;
}
```

Method 2:

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

In the 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.
Let an example, 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
unsigned int ReverseTheBits(register unsigned int x)
{
x = (((x & 0xaaaaaaaa) >> 1) | ((x & 0x55555555) << 1));
x = (((x & 0xcccccccc) >> 2) | ((x & 0x33333333) << 2));
x = (((x & 0xf0f0f0f0) >> 4) | ((x & 0x0f0f0f0f) << 4));
x = (((x & 0xff00ff00) >> 8) | ((x & 0x00ff00ff) << 8));
return((x >> 16) | (x << 16));
}
```

Third Method:

This is the simplest method to reverse the bits of an integer. In which we create a table of the hex value from 0 to 255. In this method, we are performing the AND operation of data with 0xFF to calculate the index of the array.

In this algorithm, we need to calculate the index of the array (look-up table) four times to get the appropriate value from the look-up table. After getting the

corresponding value, we will perform the bit shifting operation to get the reverse value.

```
#include <stdio.h>
#include <stdlib.h>
#define CHAR_BITS 8 // size of character
#define INT_BITS ( sizeof(int) * CHAR_BITS)
//print data in binary
void PrintInBinary(unsigned n)
{
short int iPos;
for (iPos = (INT_BITS -1) ; iPos >= 0 ; iPos--)
{
(n & (1 << iPos))? printf("1"): printf("0");
}
}
//Fastest (lookup table):
static const unsigned char TableBitReverse[] =
{
0x00, 0x80, 0x40, 0xC0, 0x20, 0xA0, 0x60, 0xE0, 0x10, 0x90, 0x50, 0xD0, 0x30, 0xB0, 0x70, 0xF0,
0x08, 0x88, 0x48, 0xC8, 0x28, 0xA8, 0x68, 0xE8, 0x18, 0x98, 0x58, 0xD8, 0x38, 0xB8, 0x78, 0xF8,
0x04, 0x84, 0x44, 0xC4, 0x24, 0xA4, 0x64, 0xE4, 0x14, 0x94, 0x54, 0xD4, 0x34, 0xB4, 0x74, 0xF4,
0x0C, 0x8C, 0x4C, 0xCC, 0x2C, 0xAC, 0x6C, 0xEC, 0x1C, 0x9C, 0x5C, 0xDC, 0x3C, 0xBC, 0x7C, 0xFC,
0x02, 0x82, 0x42, 0xC2, 0x22, 0xA2, 0x62, 0xE2, 0x12, 0x92, 0x52, 0xD2, 0x32, 0xB2, 0x72, 0xF2,
0x0A, 0x8A, 0x4A, 0xCA, 0x2A, 0xAA, 0x6A, 0xEA, 0x1A, 0x9A, 0x5A, 0xDA, 0x3A, 0xBA, 0x7A, 0xFA,
0x06, 0x86, 0x46, 0xC6, 0x26, 0xA6, 0x66, 0xE6, 0x16, 0x96, 0x56, 0xD6, 0x36, 0xB6, 0x76, 0xF6,
0x0E, 0x8E, 0x4E, 0xCE, 0x2E, 0xAE, 0x6E, 0xEE, 0x1E, 0x9E, 0x5E, 0xDE, 0x3E, 0xBE, 0x7E, 0xFE,
0x01, 0x81, 0x41, 0xC1, 0x21, 0xA1, 0x61, 0xE1, 0x11, 0x91, 0x51, 0xD1, 0x31, 0xB1, 0x71, 0xF1,
0x09, 0x89, 0x49, 0xC9, 0x29, 0xA9, 0x69, 0xE9, 0x19, 0x99, 0x59, 0xD9, 0x39, 0xB9, 0x79, 0xF9,
0x05, 0x85, 0x45, 0xC5, 0x25, 0xA5, 0x65, 0xE5, 0x15, 0x95, 0x55, 0xD5, 0x35, 0xB5, 0x75, 0xF5,
0x0D, 0x8D, 0x4D, 0xCD, 0x2D, 0xAD, 0x6D, 0xED, 0x1D, 0x9D, 0x5D, 0xDD, 0x3D, 0xBD, 0x7D, 0xFD,
0x03, 0x83, 0x43, 0xC3, 0x23, 0xA3, 0x63, 0xE3, 0x13, 0x93, 0x53, 0xD3, 0x33, 0xB3, 0x73, 0xF3,
0x0B, 0x8B, 0x4B, 0xCB, 0x2B, 0xAB, 0x6B, 0xEB, 0x1B, 0x9B, 0x5B, 0xDB, 0x3B, 0xBB, 0x7B, 0xFB,
0x07, 0x87, 0x47, 0xC7, 0x27, 0xA7, 0x67, 0xE7, 0x17, 0x97, 0x57, 0xD7, 0x37, 0xB7, 0x77, 0xF7,
0x0F, 0x8F, 0x4F, 0xCF, 0x2F, 0xAF, 0x6F, 0xEF, 0x1F, 0x9F, 0x5F, 0xDF, 0x3F, 0xBF, 0x7F, 0xFF
};
int main()
{
```

```
unsigned int data = 0;
unsigned int Ret = 0;
printf("Enter the number : ");
scanf("%u",&data);
printf("\n\nEntered Data is " );
PrintInBinary(data);
//Getting reverse value
Ret = (TableBitReverse[data & 0xff] << 24) |
(TableBitReverse[(data >> 8) & 0xff] << 16) |
(TableBitReverse[(data >> 16) & 0xff] << 8) |
(TableBitReverse[(data >> 24) & 0xff]);
printf("\n\nReverse Data is " );
PrintInBinary(Ret);
return 0;
}
```

Q) How to print a decimal number in binary format in C?

Ans:

```
#define CHAR_BITS 8 // size of character
#define INT_BITS ( sizeof(int) * CHAR_BITS) //bits in integer
void PrintInBinary(unsigned n)
{
char Pos = (INT_BITS -1);
for (; Pos >= 0 ; --Pos)
{
(n & (1 << Pos))? printf("1"): printf("0");
}
}
```

Q) What is the output of the below program?

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

```
int data = 16;
data = data >> 1;
printf("%d\n", data );
return 0;
}
```

Ans:

8

Q) What is the output of the below program?

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

Ans:

15

Q) What is the output of the below program?

```
#include <stdio.h>
#include<stdlib.h>
int main()
{
void *pvBuffer = NULL;
pvBuffer = malloc(sizeof(int));
*((int*)pvBuffer) = 0x00000000;
*((int*)pvBuffer)|= 2;
printf("OutPut = %d",*((int*)pvBuffer));
```

```
free(pvBuffer);
return 0;
}
```

Ans:

2

Q) Write a program swap two numbers without using the third variable?

Ans:

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

Method 1((Using Arithmetic Operators):

```
#include <stdio.h>
int main()
{
int a = 10, b = 5;
// algo to swap 'a' and 'b'
a = a + b; // a becomes 15
b = a - b; // b becomes 10
a = a - b; // fonally a becomes 5
printf("After Swapping the value of: a = %d, b = %d\n\n", a, b);
return 0;
}
```

Method 2 (Using Bitwise XOR Operator):

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

```
{
int a = 10, b = 5;
// algo to swap 'a' and 'b'
a = a ^ b; // a becomes (a ^ b)
b = a ^ b; // b = (a ^ b ^ b), b becomes a
a = a ^ b; // a = (a ^ b ^ a), a becomes b
printf("After Swapping the value of: a = %d, b = %d\n\n", a, b);
return 0;
}
```

Q) Write a program to check an integer is a power of 2?

Ans:

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

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

or

```
int CheckPowerOftwo (unsigned int x)
{
return (x && (!(x & (x - 1))));
}
```

Q) What is the output of the below code?

```
#include <stdio.h>
int main()
{
int x = -30;
```

```
x = x << 1;
printf("%d\n", x);
}
```

Ans:

undefined behavior.

Q) What is the output of the below code?

```
#include <stdio.h>
int main()
{
int x = -30;
x = x >> 1;
printf("%d\n", x);
return 0;
}
```

Ans:

implementation-defined.

Q) Write a program to count set bits in an integer?

Ans:

```
unsigned int NumberSetBits(unsigned int n)
{
unsigned int CountSetBits= 0;
while (n)
{
CountSetBits += n & 1;
```

```
n >>= 1;
}
return CountSetBits;
}
```

Q) When should we use pointers in a C program?

Answer:

- To pass a 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 a dynamic array.
- To create a call back function using the function pointer.

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

Q) What is void or generic pointers in C?

Ans:

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 type.

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;
```

Q) What is the advantage of a void pointer in C?

Ans:

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 functions we can copy the data from the source to destination.
  e.g.

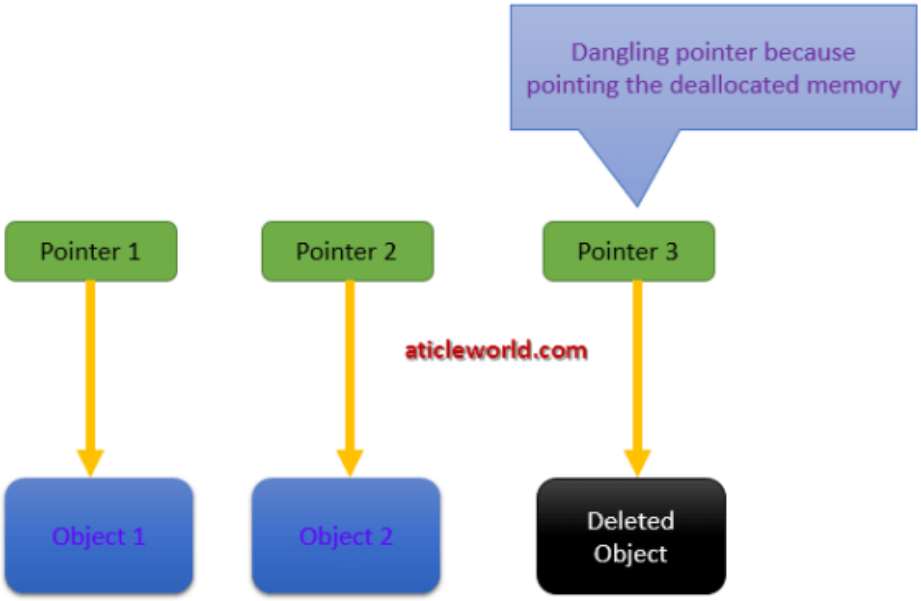
  ```
  void * memcpy ( void * dst, const void * src, size_t num );
  ```

- We have already known 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 list. For more information see this link: How to create a generic Link List.

Q) What are dangling pointers?

Ans:

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,

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
int *piData = NULL;
//creating integer of size 10.
piData = malloc(sizeof(int)* 10);
if(piData == NULL)
{
return 0;
}
//free the allocated memory
free(piData);
//piData is dangling pointer
*piData = 10;
return 0;
}
```

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.

Q) What is the wild pointer?

Ans:

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 other words, 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
```

Q) What is a NULL pointer?

Ans:

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
```

Q) What is a Function Pointer?

Ans:

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.

Q) How to declare a pointer to a function in c?

Ans:

The syntax for declaring function pointer is very straightforward. It seems 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 the 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 the declaration name is preceded by the * (Asterisk) symbol and enclosed in parenthesis.

For example,

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

For a 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 that takes the const character pointer as arguments and returns a void pointer.

Q) Where can the function pointers be used?

Ans:

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.

Q) What is the difference between array and pointer in c?

Ans:

Here is one important difference between array and pointer is the 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 the below example when trying to increment the address of the array then we will get the compiler error.

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


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

 char acBuffer [] = {'a','t','i','c' ,'l' ,'e'}; // array of character

 acBuffer++; // increment the array

 //print the element
 printf("Element of the array %d\n",*acBuffer);


return 0;
}
```

ces | Compile Log | Debug | Find Results | Close

	Message
IRA\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.

Q) What is the output of the C programming (Assumed int size is 4bytes)?

```c
#include<stdio.h>
int main()
{
int (*arr)[5][4];
```

```
//Suppose integer size 4 bytes
printf("*arr size %d", sizeof(*arr));
return 0;
}
```

Ans:

*arr size 80

Explanation:

int (*arr)[5][4] is a pointer to an array. The total number of elements the 4*5 and if integer size is 4 bytes then the size of *arr will be 80.

Q) What is static memory allocation and dynamic memory allocation?

Ans:

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 i**s 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.

Q) What is the memory leak in C?

Ans:

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.

```
int main ()
{
char * pBuffer = malloc(sizeof(char) * 20);
/* Do some work */
return 0; /*Not freeing the allocated memory*/
}
```

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

Q) What is the difference between malloc and calloc?

Ans:

A 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 takes one argument but calloc takes two.

Q) What is the purpose of realloc( )?

Ans:

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 the 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 code,

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main ()
{
char *pcBuffer = NULL;
/* Initial memory allocation */
pcBuffer = malloc(8);
if(pcBuffer == NULL)
{
```

```
return 0;
}
strcpy(pcBuffer, "aticle");
printf("pcBuffer = %s\n", pcBuffer);
/* Reallocating memory */
pcBuffer = realloc(pcBuffer, 15);
strcat(pcBuffer, "world");
printf("pcBuffer = %s\n", pcBuffer);
//free the allocated memory
free(pcBuffer);
return 0;
}
```

Output:
pcBuffer = aticle
pcBuffer = aticleworld

Note: It should be used for dynamically allocated memory but if a pointer is a null pointer, realloc behaves like the malloc function.

Q) What is the return value of malloc (0)?

Ans:

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.

Q) What is dynamic memory fragmentation?

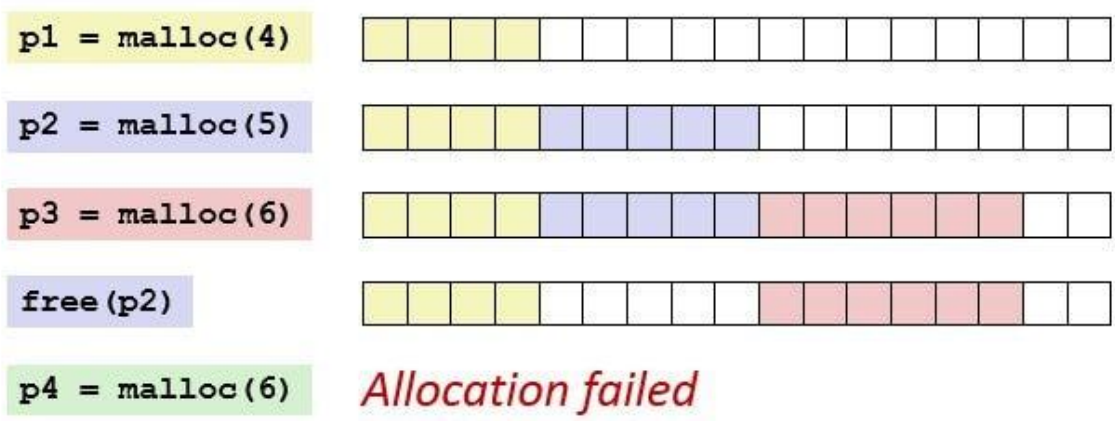
Ans:

The memory management function is guaranteed that if memory is allocated, then it would be suitably aligned to any object which has the fundamental alignment. The fundamental alignment is less than or equal to the largest alignment that's supported by the implementation without an alignment specification.

One of the major **problems with dynamic memory allocation** is fragmentation, basically, fragmentation occurred when the user does not use the memory efficiently. There are two types of fragmentation, external fragmentation, and internal fragmentation.

The external fragmentation is due to the small free blocks of memory (small memory hole) that is available on the free list but the program not able to use it. There are different types of free list allocation algorithms that used the free memory block efficiently.

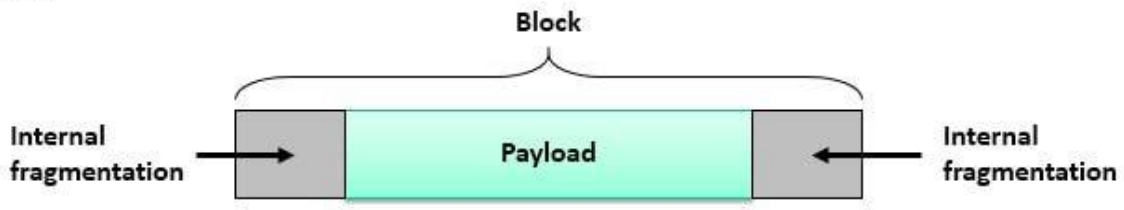
To understand the external fragmentation, consider a scenario where a program has 3 contiguous blocks of memory and the user frees the middle block of memory. In that scenario, you will not get a memory, if the required block of memory is larger than a single block of memory (but smaller or equal to the aggregate of the block of memory).



The internal fragmentation is wasted of memory that is allocated for rounding up the allocated memory and in bookkeeping (infrastructure), the bookkeeping is used to keep the information of the allocated memory.

Whenever we called the malloc function then it reserves some extra bytes (depend on implementation and system) for bookkeeping. This extra byte is reserved for each call of malloc and becomes a cause of the internal fragmentation.

If size of the payload is smaller than the block size, then internal fragmentation occur.



For example,
See the below code, the programmer may think that the system will be allocated 8 *100 (800) bytes of memory but due to bookkeeping (if 8 bytes) system will be allocated 8*100 extra bytes. This is an internal fragmentation, where 50% of the heap waste.

```
//Only sample code.
#include <stdio.h>
#include <stdlib.h>
char *acBuffer[100];
int main()
{
int iLoop = 0;
while(iLoop < 100)
{
acBuffer[iLoop ] = malloc(8);
++iLoop;
}
}
```

Q) How is the free work in C?

Ans:

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 the free function gives the undefined result.

```
___ The allocated block ___
/ \
+--------+--------------------+
| Header | Your data area ... |
+--------+--------------------+
^
|
+-- Returned Address
```

For example,

```c
#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;
}
```

Q) How can you determine the size of an allocated portion of memory?

Ans:

In C language, we can calculate the size of the static array using the sizeof operator but there is no operator to calculate the size of the dynamically allocated memory.

Mainly there are two ways to get the size of allocated memory in every section of the code.

- Create a global variable to store the size of the allocated memory.
- Carry the length of allocated memory.

For example,

Suppose you need to create an integer array whose size is n. So to carry the array length of the array, you need to allocate the memory for n+1.

```
int *piArray = malloc ( sizeof(int) * (n+1) );
```

If memory is allocated successfully, assign n (size of the array) its 0 places.

```
piArray[0] = n;
or
* piArray = n;
```

Now it's time to create a copy of the original pointer but to left one location from the beginning.

```
int * pTmpArray = piArray +1;
```

Note: if you are new, see this article arithmetic operation on the pointer.

Now, whenever in a program you ever required the size of the array then you can get from copy pointer.

```
ArraySize = pTmpArray[-1];
```

After using the allocated memory don't forget to deallocate the allocated memory.

```
free (piArray);
```

Q) What is the output of the below C code?

```
#include <stdio.h>
#include <stdlib.h>
#define ALLOC_MEMORY 5
int main()
{
int loop = 0;
int *ptr = malloc(ALLOC_MEMORY *sizeof(int));
if(ptr == NULL)
{
perror("fail to allocate memory");
return -1;
}
for(loop=0; loop < ALLOC_MEMORY; ++loop)
{
*(ptr + loop) = loop;
}
printf(" %d",*ptr++);
printf(" %d",(*ptr)++);
printf(" %d",*ptr);
printf(" %d",*++ptr);
printf(" %d",++*ptr);
free(ptr);
return 0;
}
```

Ans:

0 1 2 2 3

Explanation:

1st printf:- *ptr++ means that it will increment the address and dereference that address but here the increment is a post increment so dereference first and after increment, so on the base address you get 0 (ptr is pointing to next location).

2nd printf:- (*ptr)++ = first dereference and than increment the value so on that location value is 1 is one is increment so you get 2 (here pointer is not changed).

3rd printf:- *ptr mean when the pointer is pointing do dereference on that location so you get 2.

4th printf:- *++ptr mean the first pointer is increment after dereferencing so you get 2 (pointer is changed).

5th printf:- ++*ptr mean first dereference and than increment the value so you get as 3

Q) What is the difference between memcpy and memmove?

Ans:

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.

```
#include <string.h>
#include <stdio.h>
char str1[50] = "I am going from Delhi to Gorakhpur";
char str2[50] = "I am going from Gorakhpur to Delhi";
int main( void )
{
//Use of memmove
printf( "Function:\tmemmove with overlap\n" );
printf( "Orignal :\t%s\n",str1);
printf( "Source:\t\t%s\n", str1 + 5 );
printf( "Destination:\t%s\n", str1 + 11 );
memmove( str1 + 11, str1 + 5, 29 );
printf( "Result:\t\t%s\n", str1 );
printf( "Length:\t\t%d characters\n\n", strlen( str1 ) );
//Use of memcpy
printf( "Function:\tmemcpy with overlap\n" );
printf( "Orignal :\t%s\n",str2);
printf( "Source:\t\t%s\n", str2 + 5 );
printf( "Destination:\t%s\n", str2 + 11 );
memcpy( str2 + 11, str2 + 5, 29 );
printf( "Result:\t\t%s\n", str2 );
printf( "Length:\t\t%d characters\n\n", strlen( str2 ) );
return 0;
}
```

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 scenarios.

Q) Reverse a string in c without using library function

Ans:

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

Here we will **reverse a string** using two methods, iterative and recursive.

Reversing a string using the Iterative method

Method 1:

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

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
char acData[100]= {0}, Temp = 0;
int iLoop =0, iLen = 0;
int cnt =0;
printf("\nEnter the string :");
//Maximum takes 100 character
if (fgets(acData,100,stdin) == NULL)
{
printf("Error\n");
return 1;
}
//calculate length of string
while(acData[iLen++] != '\0');
//Remove the null character
iLen--;
//Array index start from 0 to (length -1)
iLen--;
```

```
while (iLoop < iLen)
{
Temp = acData[iLoop];
acData[iLoop] = acData[iLen];
acData[iLen] = Temp;
iLoop++;
iLen--;
}
printf("\n\nReverse string is : %s",acData);
return 0;
}
```

Method 2:

```
#include <stdio.h>
#include <stdlib.h>
#define SWAP_CHARACTER(a,b) do { \
(*a)^=(*b); \
(*b)^=(*a);\
(*a)^=(*b);\
a++; \
b--; \
}while(0);
int main(int argc, char *argv[])
{
char acData[100]= {0};
char *pcStart = NULL;
char *pcEnd = NULL;
int iLoop =0, iLen = 0;
printf("\nEnter the string :");
//Maximum takes 100 character
if (fgets(acData,100,stdin) == NULL)
{
printf("Error\n");
return 1;
}
//Pointer point to the address of first character
```

```
pcStart = acData;
// calculate length of string
while(acData[iLen++] != '\0');
//Remove the null character
iLen--;
pcEnd = (pcStart + iLen-1);
while (iLoop < iLen/2)
{
SWAP_CHARACTER (pcStart,pcEnd);
iLoop++;
}
printf("\n\nReverse string is : %s\n\n",acData);
return 0;
}
```

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

```
#include <stdio.h>
#include <stdlib.h>
//recursive function
void StringRev(char *pszInputData, unsigned int Start, unsigned int End)
{
if(Start >= End)
{
return 1;
}
// swap the data
*(pszInputData + Start) = *(pszInputData + Start) ^ *(pszInputData + End);
```

```
*(pszInputData + End) = *(pszInputData + Start) ^ *(pszInputData + End);
*(pszInputData + Start) = *(pszInputData + Start) ^ *(pszInputData + End);
//function called repeatedly
StringRev(pszInputData,Start+1, End-1);
}
int main(int argc, char *argv[])
{
char acData[100]= {0};
int iLen = 0;
unsigned int Start=0;
printf("\nEnter the string :");
//Maximum takes 100 character
if (fgets(acData,100,stdin) == NULL)
{
printf("Error\n");
return 1;
}
// calculate length of string
while(acData[iLen++] != '\0');
//Remove the null character
iLen--;
//Find array last index
iLen--;
StringRev(acData,Start, iLen);
printf("\n\nReverse string is : %s\n\n",acData);
return 0;
}
```

Q) What is the endianness?

Ans:

The endianness is the order of bytes to store data in memory and it also describes the order of byte transmission over a digital link. In the memory data store in which order depends on the endianness of the system, if the system is **big-endian** then the MSB byte store first (means at lower address) and if the system is little-endian then LSB byte store first (means at lower address).

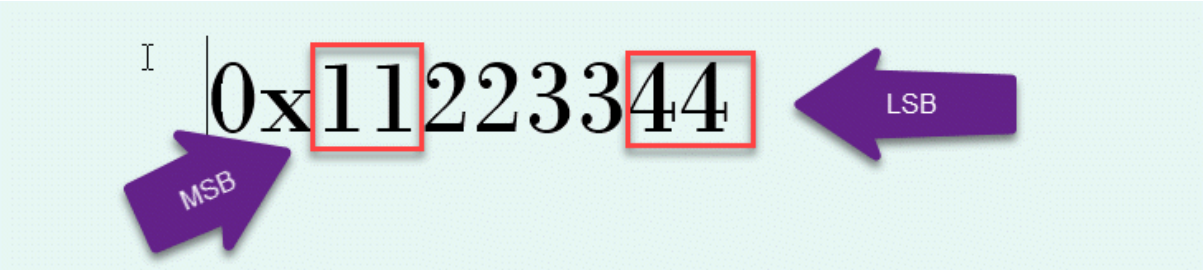
Some examples of the little-endian and big-endian system.

Little Endian	Big Endian
• Intel x86 and x86-64 series • Zilog Z80 (including Z180 and eZ80) • MOS Technology 6502	• Motorola 68000 series • Xilinx Microblaze, SuperH, • IBM z/Architecture, Atmel AVR32.

Q) What is big-endian and little-endian?

Ans:

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.

Q) Write a C program to check the endianness of the system?

Ans:

Below I am writing C programs to check the endianness of the system.

Method 1:

```
#include <stdio.h>
#include <stdlib.h>
#include <inttypes.h>
int main(void)
{
uint32_t u32RawData;
uint8_t *pu8CheckData;
u32RawData = 0x11223344; //Assign data
pu8CheckData = (uint8_t *)&u32RawData; //Type cast
```

```
if (*pu8CheckData == 0x44) //check the value of lower address
{
printf("little-endian");
}
else if (*pu8CheckData == 0x11) //check the value of lower address
{
printf("big-endian");
}
return 0;
}
```

Method 2:

```
#include <stdio.h>
#include <stdlib.h>
#include <inttypes.h>
typedef union
{
uint32_t u32RawData; // integer variable
uint8_t au8DataBuff[4]; //array of character
} RawData;
int main(void)
{
RawData uCheckEndianess;
uCheckEndianess.u32RawData = 0x11223344; //assign the value
//check the array first index value
if (uCheckEndianess.au8DataBuff[0] == 0x44)
{
printf("little-endian");
} //check the array first index value
else if (uCheckEndianess.au8DataBuff[0] == 0x11)
{
printf("big-endian");
}
return 0;
}
```

Q) How to Convert little-endian to big-endian vice versa in C?

Ans:

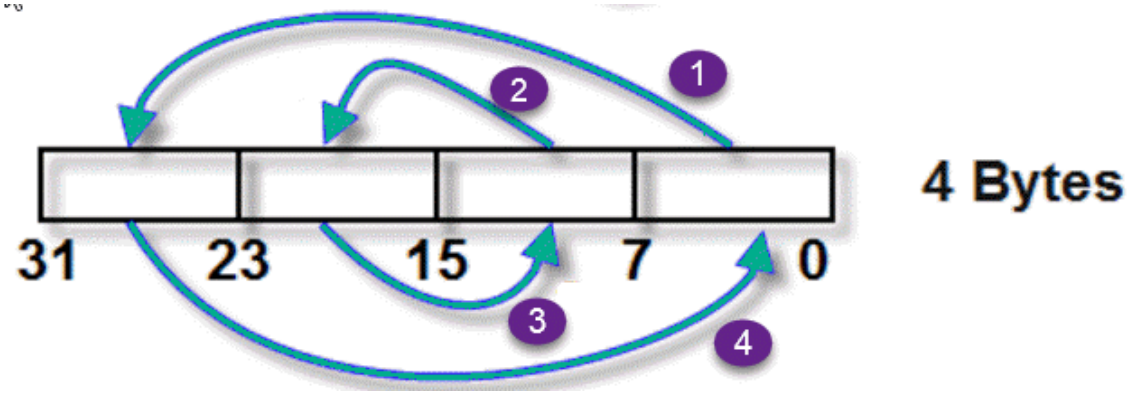
Below I am writing C programs to convert little-endian to big-endian (vice versa).

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;
}
```

Output:

```
0x44332211
0x11223344
```



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;
}
```

Q) Write a c Program to Check Whether a Number is Prime or Not?

Ans:

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

You can see Article, How to find all prime numbers up to n.

Algorithm to check prime number using division method

START

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

STOP

```
#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;
}
```

Output:

```
Enter the number : 11
11 is prime number..
```

Q) How to find the size of an array in C without using the sizeof operator?

Ans:

Method 1:

```
#include <stdio.h>
int main(int argc, char *argv[])
{
int iTotalElement = 0 ;
int aiData[] = {10, 20, 30, 40, 50, 60};
//Calculate numbers of elements using pointer arithmatic
iTotalElement = *(&aiData + 1) - aiData;
printf("Number of element = %d",iTotalElement);
return 0;
}
```

Output:

```
Number of element = 6
```

Method 2:

```
#include <stdio.h>
// User created size of operator
#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;
}
```

Output:

```
Number of element = 6
```

Q) How to find the size of the structure in c without using sizeof operator?

Ans:

Method 1:

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

- First, create the structure.
- Create a pointer to structure and assign the NULL pointer.
- 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);
return 0;
}
```

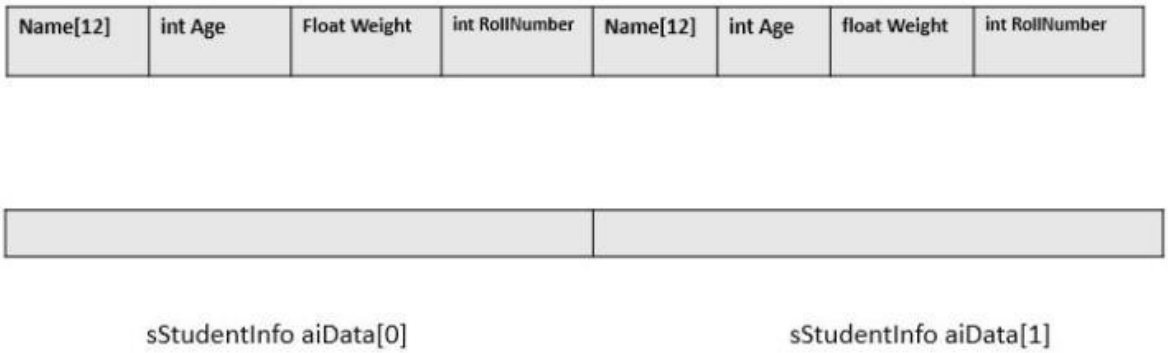
Output:

```
Size of structure = 24
```

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.

- First, create the structure.
- create an array of structure, Here aiData[2].
- Create pointers to the structure and assign the address of the first and second element of the array.
- 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;
}
```

Output:

```
Size of structure = 24
```

Q) What is meant by structure padding?

Ans:

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 the penalty of performance.

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

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

Ans:

In C language, there are a lot of ways to pass the 2d array as a parameter. In the 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>
//Size of the created array
#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[])
{
```

```
//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 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>
//Size of the created array
#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;
}
```

Q) What is the difference between enum and macro?

Ans:

- An enum increases the readability of the code and easy to debug in comparison to the macro.
- All elements of enum grouped together which is not possible with macro.

for example,

```
//constant created by macro,
#define MON 0
#define TUE 1
#define WED 2
#define THU 3
#define FRI 4
#define SAT 5
#define SUN 6
//constant created by enum,
typedef enum Days
{
Mon,
Tue,
Wed,
Thu,
Fri,
Sat,
Sun
} Weekday;
```

- enum defines a new type but the macro does not define a new type.
- enum follow scope rules and the compiler automatically assigns the value to its member constant.
- the enum type is an integer but the macro type can be any type.

Q) Delete a node in a linked list without a head pointer?

There is no practical solution to delete a node directly by given pointer, we need to do some trick. We need to copy the data from the next node to the current node by given **pointer** to be deleted and delete the next node.

```
//Get the Address of the next node
NodePointer temp = Node->pNextNode;
//Get the data of next node
Node->iData = temp->iData;
//Get the Address of next to next node
Node->pNextNode = temp->pNextNode;
//Delete the next node
free(temp);
```

For more detail see this article, Delete a node in linked list without head pointer

Q) How do you define a multiline macro in C?

See the below example in which I am swapping the value of two variables.

```
#include <stdio.h>
#define swap(x,y,T) do { \
T temp = (*x);\
(*x) = (*y); \
(*y) = temp; \
} while (0)
int main(void)
{
int a = 5;
int b = 9;
printf("Value of a and b before swaping\n");
printf("a = %d\n",a);
printf("b = %d\n",b);
```

```
//Swap the number
swap(&a,&b,int);
printf("\n\nValue of a and b After swaping\n");
printf("a = %d\n",a);
printf("b = %d\n",b);
return 0;
}
```

Output:

```
Value of a and b before swaping
a = 5
b = 9


Value of a and b After swaping
a = 9
b = 5
```

Q) What is recursion in C?

Recursion is a process in which function call itself and the function that calls itself directly or indirectly called a recursive function. A recursive function calls itself so there can be several numbers of the recursive call, so the recursive function should have the termination condition to break the recursion. If there is a not termination condition in a recursive function, a stack overflow will occur and your program will crash.

Recursive functions are useful to solve many mathematical problems, like generating **Fibonacci series**, calculating **factorial of a number** and convenient for recursively defined data structures like trees.

Let's see an example,

```
void test( int n)
{
test(n);
// Remaining code
}
```

In the above code, the test() is a recursive function that calls itself. You can see, if you will not put termination condition in the test(), stack-overflow will occur. So we have to put a condition in test() to avoid an infinite **loop** (until not stack overflow occurs).

See the below code,

```
void test()
{
if(condition)
test();
//code
}
```

for more detail see this article, recursion in c

Q) Explain the memory Layout of the C program.

Ans:

Basically, the memory layout of the C program contains five segments these are the stack segment, heap segment, BSS (block started by symbol), DS (Data Segment) and text segment.

Each segment has own read, write and executable permission. If a program tries to access the memory in a way that is not allowed then **segmentation fault** occurs.

Below find the memory Layout of C Program

1. Stack
2. Heap
3. BSS (Uninitialized data segment)
4. DS (Initialized data segment)
5. Text

For more detail see this article, **Memory Layout of C Program.**

Q) What is the difference between array_name and &array_name?

Ans:

To understand this question let's take an example, suppose arris 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 &arrwill be similar to int(*)[5];

```
#include<stdio.h>
int main()
{
int arr[5] = {0};
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 &arris same but the output of arr+1 and &arr+1 will be not the same due to the different pointer type.

```
arr= 6356732

&arr= 6356732

arr+1 = 6356736

&arr+1 = 6356752
```

Q) C Program to Find Largest Number Among "N" Numbers without using Array?

Ans:

```
#include<stdio.h>
int main()
{
int count = 0;
int numb1 = 0;
int numb2 =0;
int i =0;
printf("Enter count of numbers = ");
scanf("%d",&count);
if(count <= 0)
{
printf("Enter valid count\n");
return 0;
}
printf("Enter the number = ");
scanf("%d",&numb1);
//loop to get next number
for(i =1; i< count ; ++i)
{
printf("Enter the number = ");
scanf("%d",&numb2);
if(numb1 < numb2)
{
numb1 = numb2;
}
}
printf("Largest number is = %d\n",numb1);
return 0;
}
```

Output:

```
Enter count of numbers = 4
Enter the number = 12
Enter the number = 1284
Enter the number = -887
Enter the number = 9
Largest number is = 1284
```

Q) What is the use of a double pointer (pointer to pointer) in C?

Answer:

There is a lot of application of double-pointer in C language but here I am describing one important application of double-pointer. If you want to create a function to allocate the memory and you want to get back the allocated memory from the function parameter, then you need to use the double-pointer in that scenario. See the below code,

```
#include<stdio.h>
#include<stdlib.h>
void AllocateMemory(int **pGetMemory,int n)
{
int *p = malloc(sizeof(int)*n);
if(p == NULL)
{
*pGetMemory = NULL;
}
else
{
*pGetMemory = p;
}
}
int main()
{
int *arr = NULL;
```

```
int len = 10;
int i =0;
//Allocate the memory
AllocateMemory(&arr,len);
if(arr == NULL)
{
printf("Failed to allocate the memory\n");
return -1;
}
//Store the value
for(i=0; i<len; i++)
{
arr[i] = i;
}
//print the value
for(i=0; i<len; i++)
{
printf("arr[%d] = %d\n",i, arr[i]);
}
//free the memory
free(arr);
return 0;
}
```

Output:

```
arr[0] = 0
arr[1] = 1
arr[2] = 2
arr[3] = 3
arr[4] = 4
arr[5] = 5
arr[6] = 6
arr[7] = 7
arr[8] = 8
arr[9] = 9
```

Q) What is the output of the below C program?

```
#include <stdio.h>
int main()
{
int a = 0;
while(a < printf("HI"))
{
++a;
}
return 0;
}
```

Ans:

The code will print “HI” three times. The printf function will return the number of characters it is printing, and this will be compared with “a”. As the return value of printf is 2, HI will be printed 2times. At last, iteration when the value of “a” is 3, first it prints HI and checks for the condition and comes out of while loop as the condition gets failed. So, HI will be printed 3 times.

Q) What is the output of the below C program?

```
#include <stdio.h>
#include <string.h>
int main(void)
{
char str[] = "#Aticleworld";
const int ret = strcmp(str,"#aticleworld");
printf("ret = %d\n",ret);
return 0;
}
```

Output:

A negative value.

Explanation:

int strcmp(const char *s1, const char *s2);
strcmp() returns an integer indicating the result of the comparison,
as follows:
0, if the s1 and s2 are equal;
A negative value if s1 is less than s2;
A positive value if s1 is greater than s2;

Q) What is the output of the below C program?

```
#include <stdio.h>
int print();
int main()
{
printf("%d\n", print());
return 0;
}
int print()
{
printf("hello");
}
```

Output:

Undefined.

Explanation:

Reaching the end of a function other than main is equivalent to return; Reaching the end of any other value-returning function is undefined behavior, but only if the result of the function is used in an expression. Executing the return statement in a no-return function is undefined behavior.

Read this article for more detail: **return statement in C.**