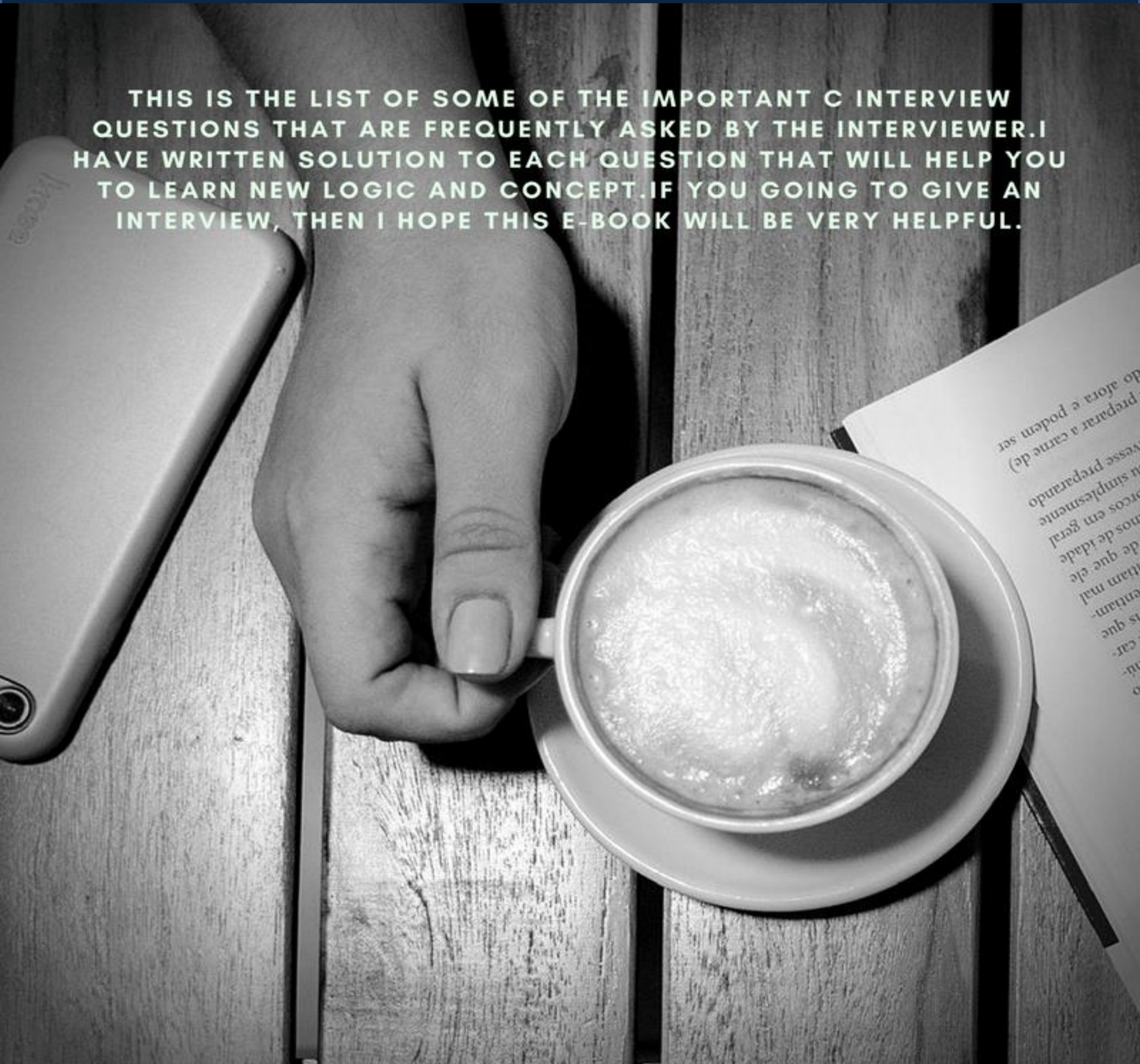
Aticleworld

C Interview Questions

www.aticleworld.com

Overview


THIS IS THE LIST OF SOME OF THE IMPORTANT C INTERVIEW QUESTIONS THAT ARE FREQUENTLY ASKED BY THE INTERVIEWER.I HAVE WRITTEN SOLUTION TO EACH QUESTION THAT WILL HELP YOU TO LEARN NEW LOGIC AND CONCEPT.IF YOU GOING TO GIVE AN INTERVIEW, THEN I HOPE THIS E-BOOK WILL BE VERY HELPFUL.

Question 1: 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 2: 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 3: 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 4: 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 5: 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 6: 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 7: 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 8: What is the difference between const and macro?

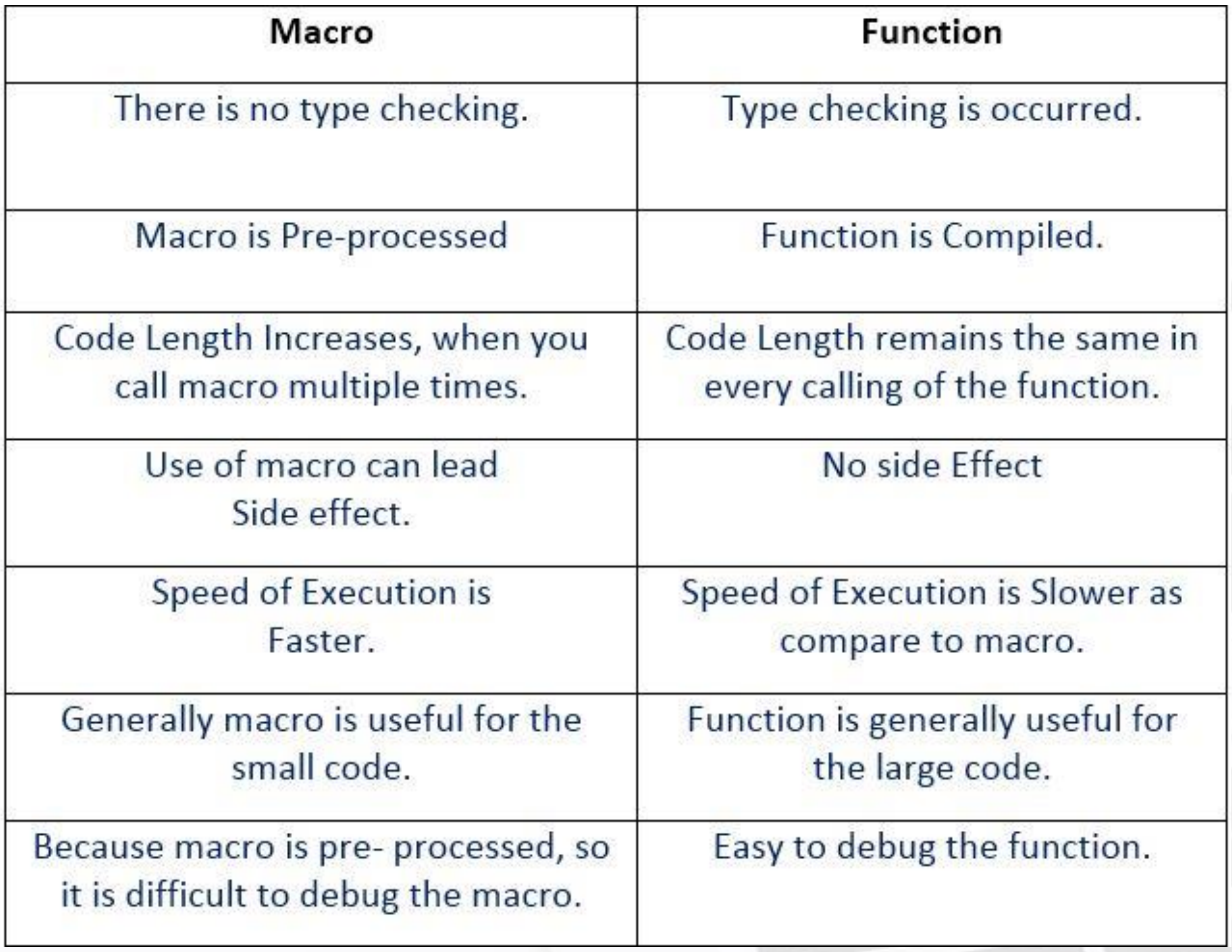
1. The const keyword is handled by the compiler, in another hand, a macro is handled by the preprocessor directive.
2. const is a qualifier that is modified the behavior of the identifier but macro is preprocessor directive.
3. There is type checking is occurred with const keyword but does not occur 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 9: 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 10: 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 11: 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 12: 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.

```
#include <stdio.h>

int main(void)
{

 char acData[5] ={'A','B','C','D','E'};
 char *pcData = NULL;

 pcData = acData;

printf("%c ",*++pcData);

return 0;
}
```

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;
}
```

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.

```
iLoop = 0.

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

Question 13: 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 13: 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);

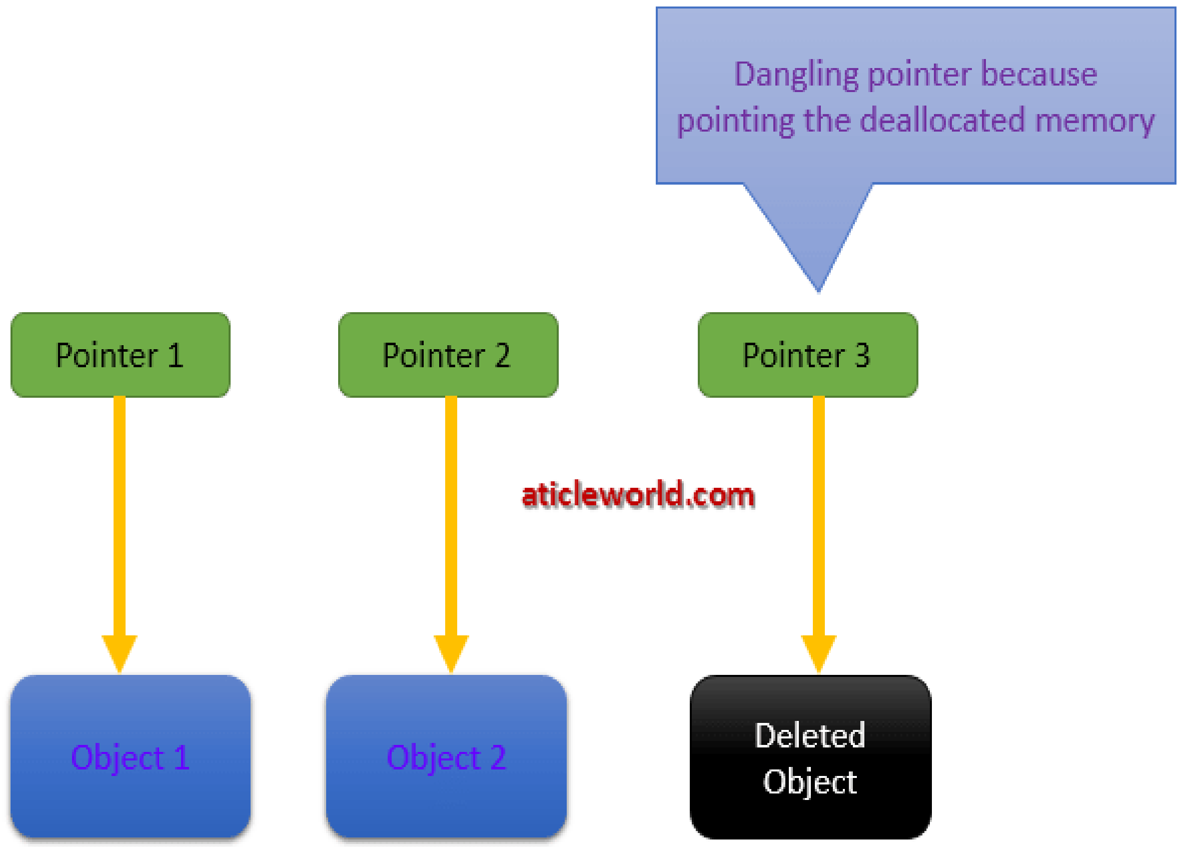
- We have already know that void pointer can be converted to another data type that is the reason malloc, calloc or realloc library function return void *. Due to the void * these functions are used to allocate memory to any data type.

- Using the void * we can create a generic linked list.

Question 14: 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.

See the below image,



For example,

```
#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

*piData = 10; //piData is dangling pointer

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.

Question 15: 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.

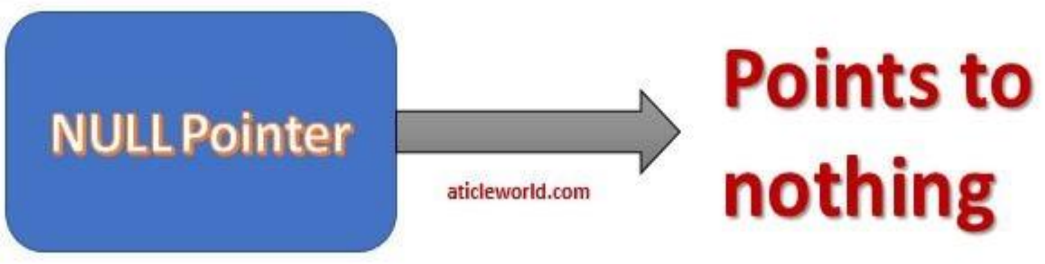
Note: Generally, compilers warn about the wild pointer.

Syntax,

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

Question 16: 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,

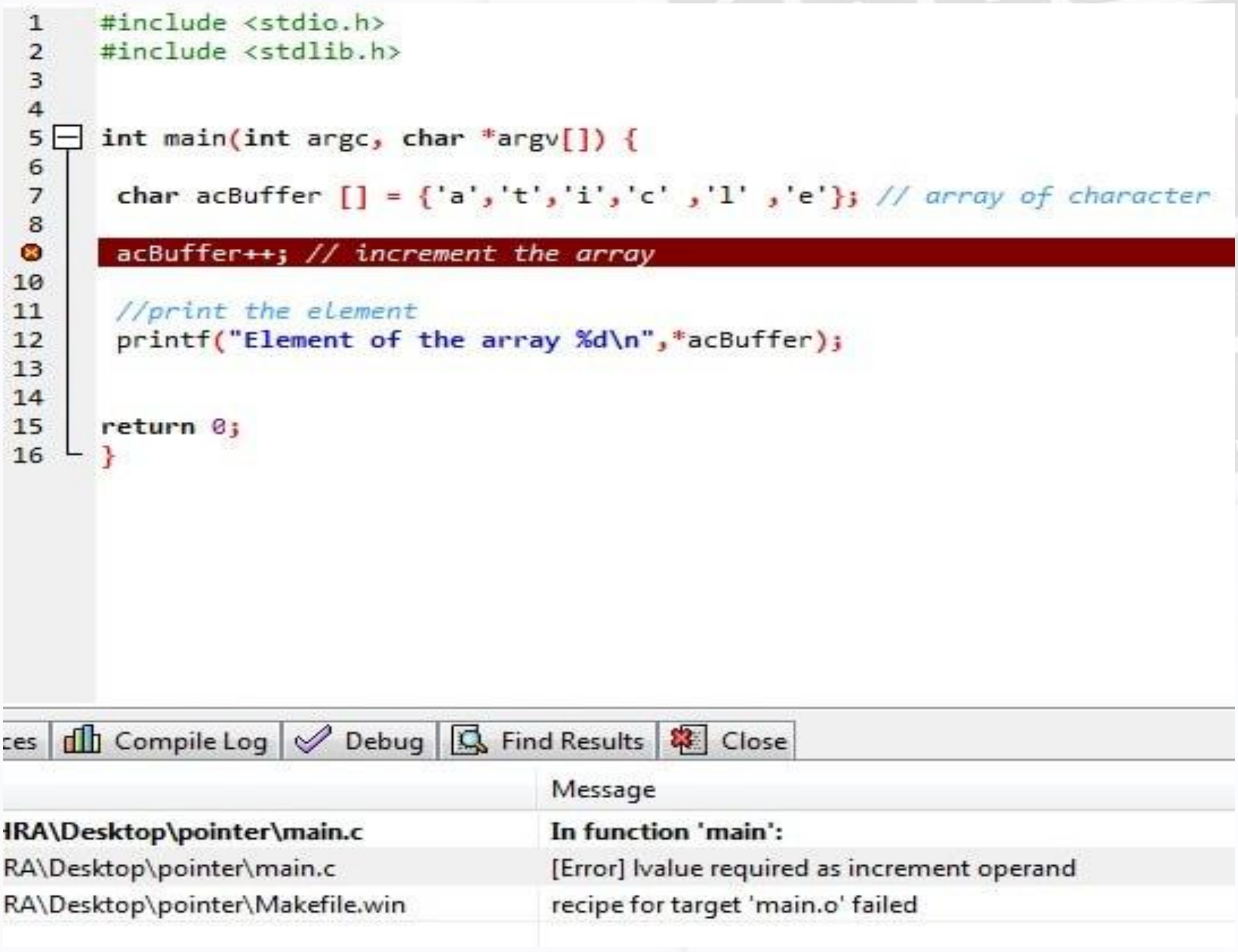
```
int *piData = NULL; // piData is a null pointer
```

Question 17: 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.



```
#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;
}
```

```
                                       Message
ıRA\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 18: 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 19: 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.

```
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.

Question 20: 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 21: 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 22: 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.

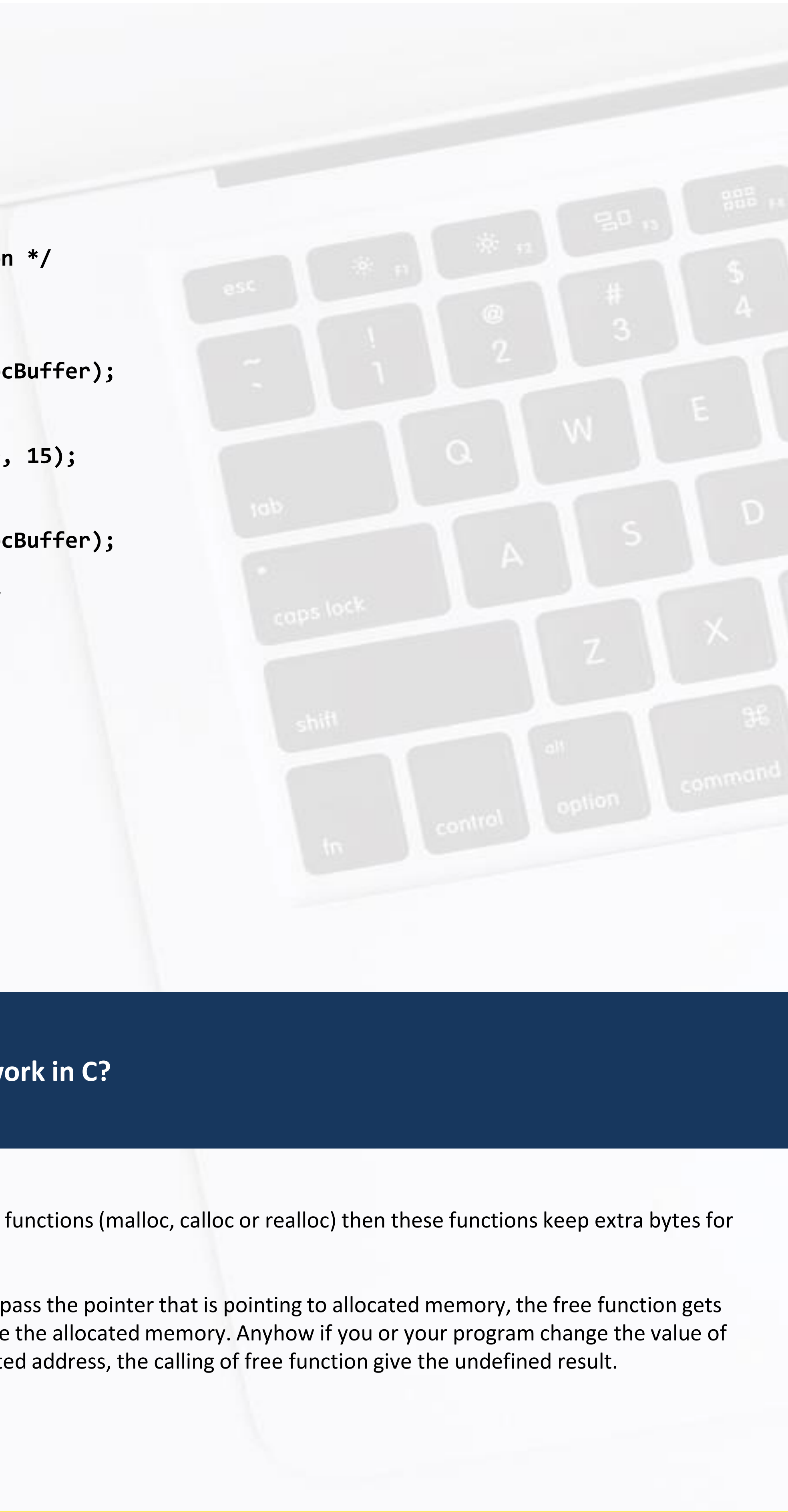
The calloc function first deallocates the old object and allocates again with newly specified size. If the new size is lesser to the old size, the contents of the newly allocated memory will be same as prior but if any bytes in the newly created object goes beyond the old size, the values of the exceeded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

Syntax:

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

Example,

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

int main ()
{

char *pcBuffer = NULL;

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

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;
}
```



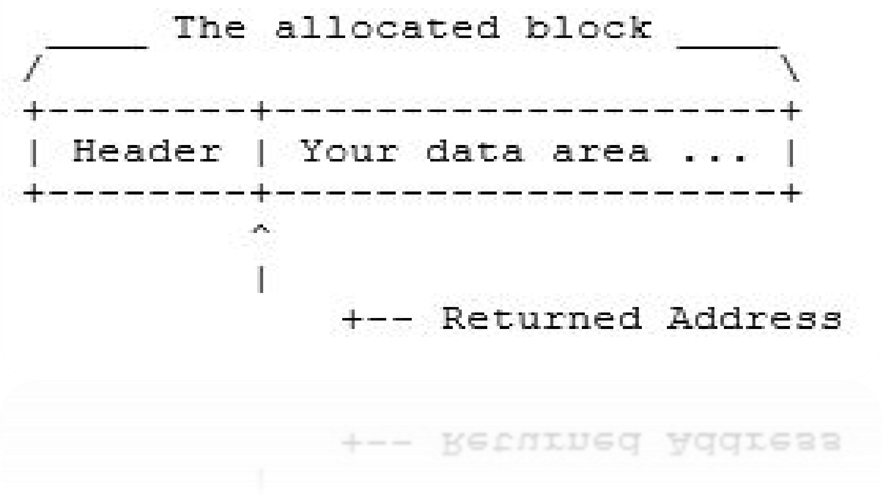
Output:

```
pcBuffer = aticle
pcBuffer = aticleworld
```

Question 23: 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.



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

For example,

```
#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;
}
```

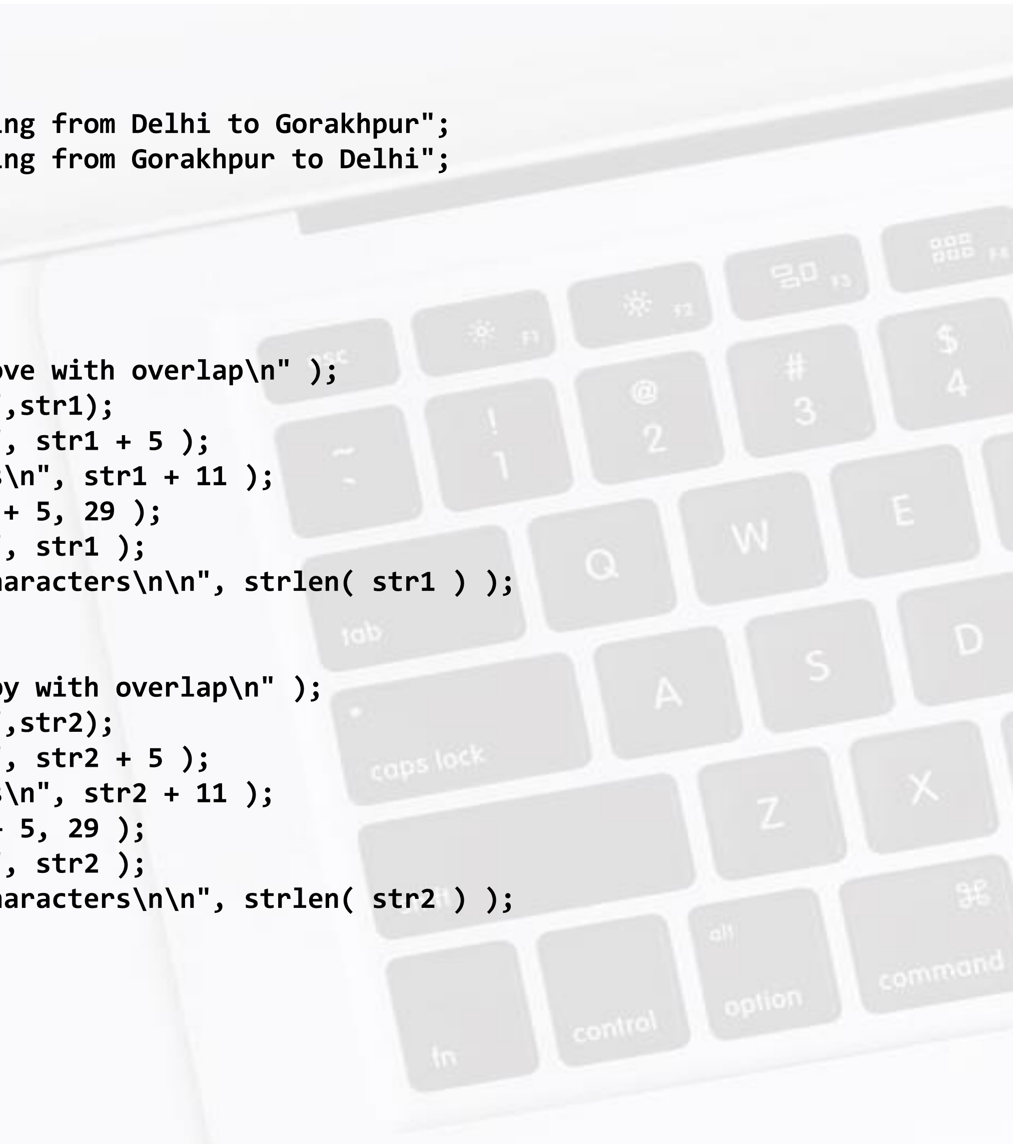
Question 24: What is the difference between memcpy and memmove?

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,

```
#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;
}
```



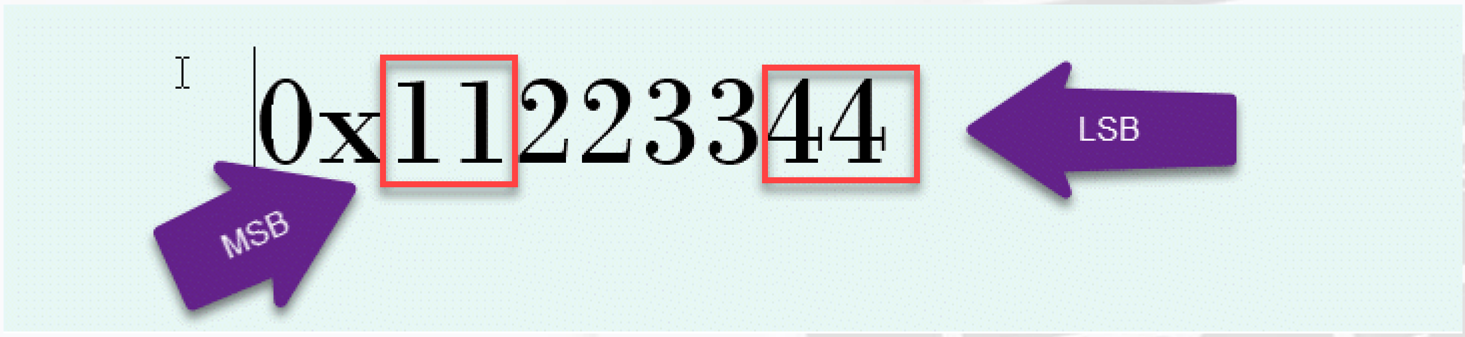
OutPut:

```
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 scenario.

Question 25: 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.

Get the full version of this eBook with 100 interview questions and Live instructor support

Download Now

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