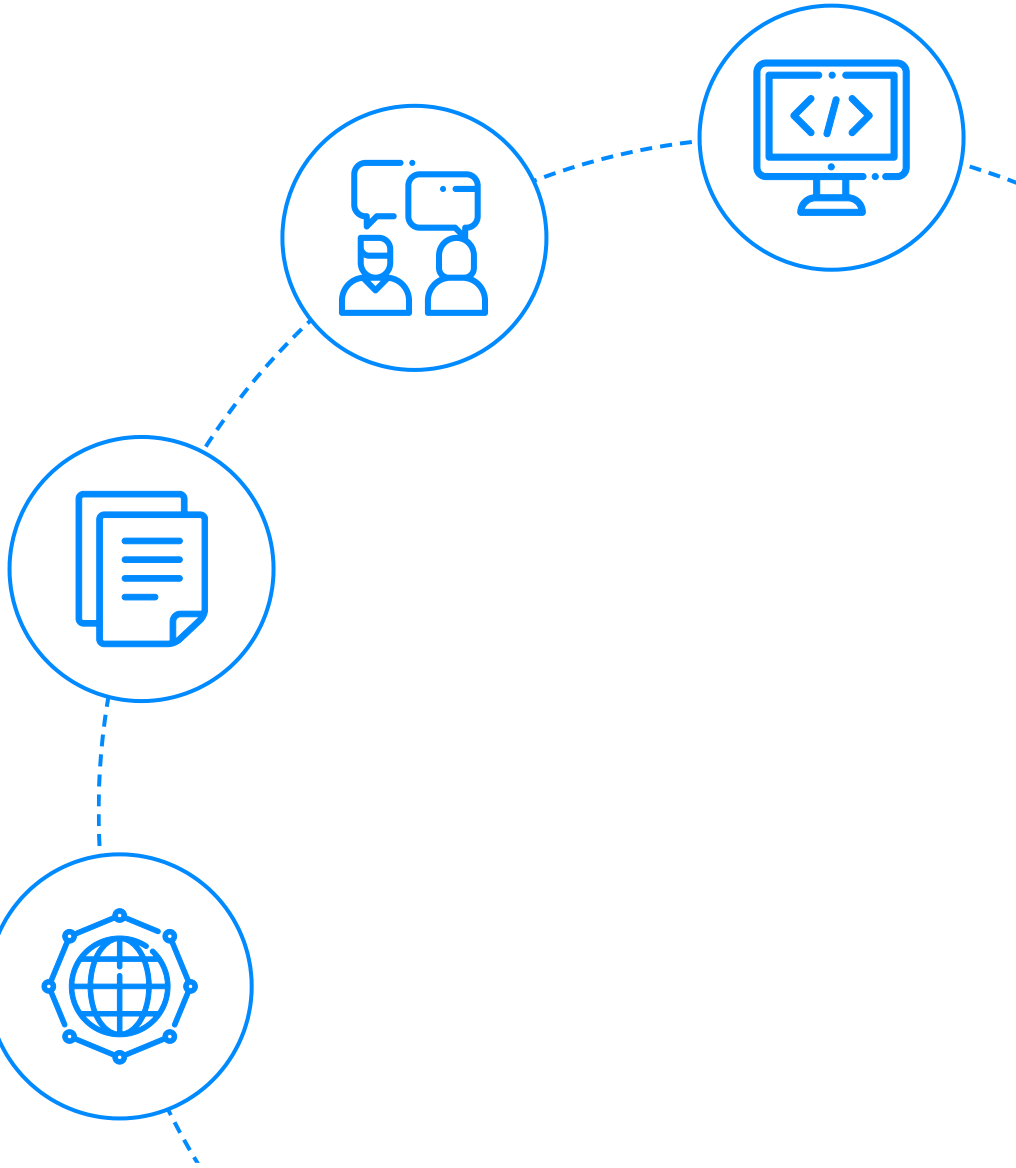
InterviewBit

Embedded C Interview Questions



To view the live version of the page, click here.

Contents

Embedded C Interview Questions for Freshers

Embedded C Interview Questions for Experienced

Embedded C Interview Questions for Experienced

(.....Continued)

19. Why is the statement ++i faster than i+1?
20. What are the reasons for segmentation fault in Embedded C? How do you avoid these errors?
21. Is it recommended to use printf() inside ISR?
22. Is it possible to pass a parameter to ISR or return a value from it?
23. What Is a Virtual Memory in Embedded C and how can it be implemented?
24. What is the issue with the following piece of code?
25. The following piece of code uses __interrupt keyword to define an ISR. Comment on the correctness of the code.
26. What is the result of the below code?
27. What are the reasons for Interrupt Latency and how to reduce it?
28. Is it possible to protect a character pointer from accidentally pointing it to a different address?
29. What do you understand by Wild Pointer? How is it different from Dangling Pointer?
30. What are the differences between the following 2 statements #include "..." and #include <...>?
31. When does a memory leak occur? What are the ways of avoiding it?

Embedded C Programming

32. Write an Embedded C program to multiply any number by 9 in the fastest manner.
33. How will you swap two variables? Write different approaches for the same.
34. Write a program to check if a number is a power of 2 or not.
35. Write a program to print numbers from 1 to 100 without making use of conditional operators?
36. Write a MIN macro program that takes two arguments and returns the smallest of both arguments.

Let's get Started

Embedded C is the most popular choice of language used for developing embedded systems because of its simplicity, efficiency, less time required for development, and its portability from one system to another. As we know that the embedded systems have constraints on hardware resources such as CPU, memory sizes, etc, it becomes very important to use the resources judiciously and responsibly. To achieve this, Embedded C usually can interact with the hardware resources with necessary abstractions.

In this article, we will see the most commonly asked interview questions in Embedded C for both freshers and experienced developers.

Embedded C Interview Questions for Freshers

1. What is Embedded C Programming? How is Embedded C different from C language?

Embedded C is a programming language that is an extension of C programming. It uses the same syntax as C and it is called “embedded” because it is used widely in embedded systems. Embedded C supports I/O hardware operations and addressing, fixed-point arithmetic operations, memory/address space access, and various other features that are required to develop fool-proof embedded systems.

Following are the differences between traditional C language and Embedded C:

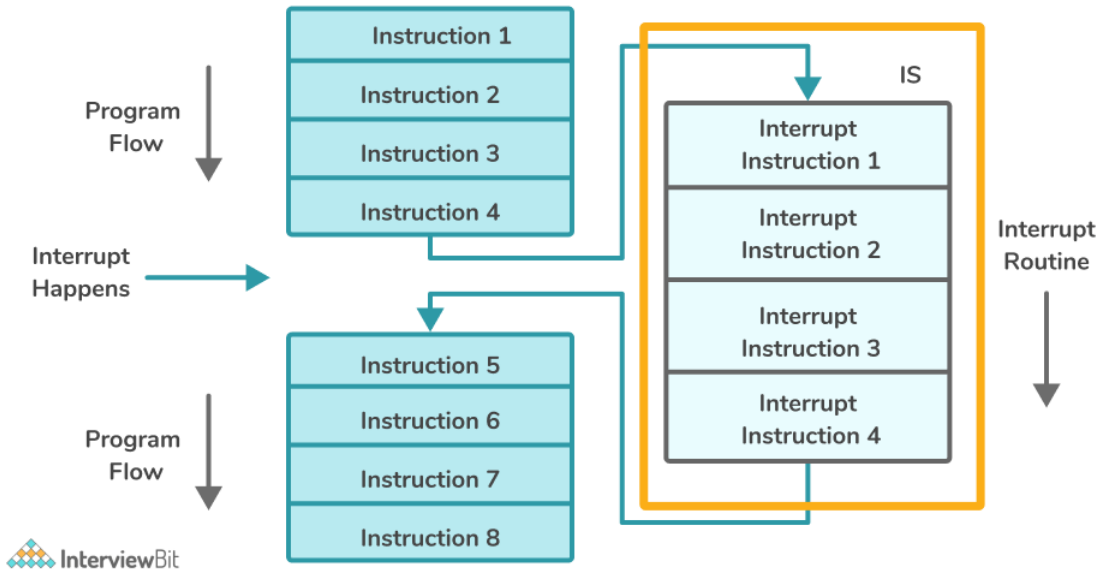
C Language	Embedded C Language
It is of native development nature	It is used for cross-development purposes
C is independent of hardware and its underlying architecture	Embedded C is dependent on the hardware architecture.
C is mainly used for developing desktop applications.	Embedded C is used in embedded systems that have limited resources like ROM, RAM, etc.

2. What do you understand by startup code?

A startup code is that piece of code that is called before the execution of the main function. This is used for creating a basic platform for the application and it is written in assembly language.

3. What is ISR?

ISR expands to Interrupt Service Routines. These are the procedures stored at a particular memory location and are called when certain interrupts occur. Interrupt refers to the signal sent to the processor that indicates there is a high-priority event that requires immediate attention. The processor suspends the normal flow of the program, executes the instructions in ISR to cater for the high priority event. Post execution of the ISR, the normal flow of the program resumes. The following diagrams represent the flow of ISR.



4. What is Void Pointer in Embedded C and why is it used?

Void pointers are those pointers that point to a variable of any type. It is a generic pointer as it is not dependent on any of the inbuilt or user-defined data types while referencing. During dereferencing of the pointer, we require the correct data type to which the data needs to be dereferenced.

For Example:

```
int num1 = 20;    //variable of int datatype
void *ptr;        //Void Pointer
*ptr = &num1;    //Point the pointer to int data
print("%d",(*(int*)ptr));    //Dereferencing requires specific data type

char c = 'a';
*ptr = &c;    //Same void pointer can be used to point to data of different type -> reu
print("%c",(*(char*)ptr));
```

Void pointers are used mainly because of their nature of re-usability. It is reusable because any type of data can be stored.

5. Why do we use the volatile keyword?

The `volatile` keyword is mainly used for preventing a compiler from optimizing a variable that might change its behaviour unexpectedly post the optimization. Consider a scenario where we have a variable where there is a possibility of its value getting updated by some event or a signal, then we need to tell the compiler not to optimize it and load that variable every time it is called. To inform the compiler, we use the keyword volatile at the time of variable declaration.

```
// Declaring volatile variable - SYNTAX
// volatile datatype variable_name;
volatile int x;
```

Here, x is an integer variable that is defined as a volatile variable.

6. What are the differences between the const and volatile qualifiers in embedded C?

const	volatile
The keyword “const” is enforced by the compiler and tells it that no changes can be made to the value of that object/variable during program execution.	The keyword “volatile” tells the compiler to not perform any optimization on the variables and not to assume anything about the variables against which it is declared.
Example: `const int x=20;` , here if the program attempts to modify the value of x, then there would be a compiler error as there is const keyword assigned which makes the variable x non-modifiable.	Example: `volatile int x;` , here the compiler is told to not assume anything regarding the variable x and avoid performing optimizations on it. Every time the compiler encounters the variable, fetch it from the memory it is assigned to.

7. What Is Concatenation Operator in Embedded C?

The Concatenation operator is indicated by the usage of `##` . It is used in macros to perform concatenation of the arguments in the macro. We need to keep note that only the arguments are concatenated, not the values of those arguments.

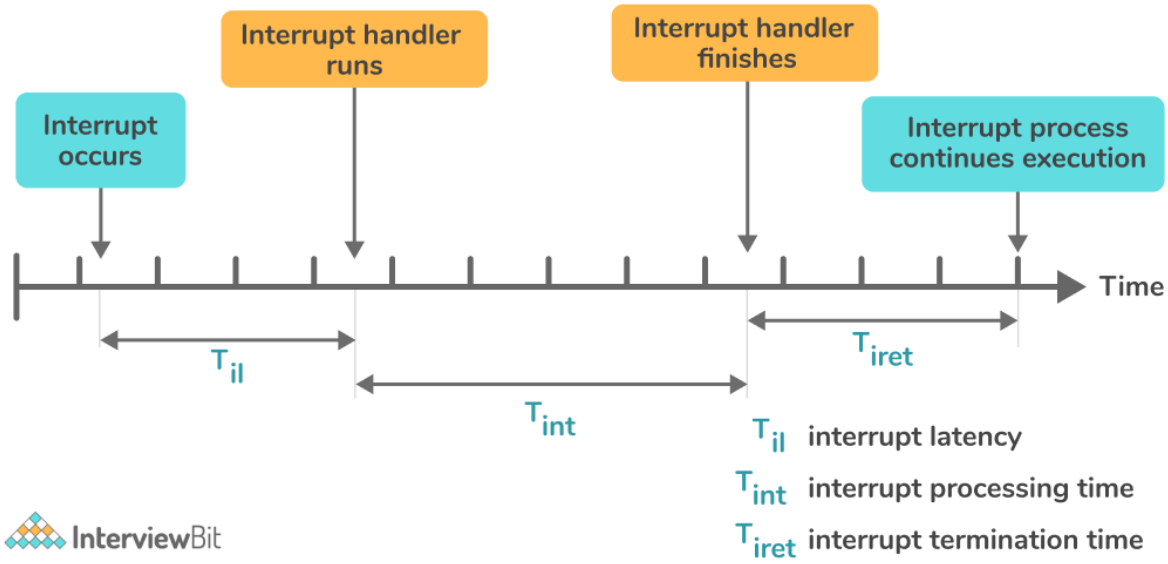
For example, if we have the following piece of code:

```
#define CUSTOM_MACRO(x, y) x##y

main(){

  int xValue = 20;
  printf("%d", CUSTOM_MACRO(x, Value));    //Prints 20

}
```

We can think of it like this if arguments `x` and `y` are passed, then the macro just returns `xy` -> The concatenation of `x` and `y`.

8. What do you understand by Interrupt Latency?

Interrupt latency refers to the time taken by ISR to respond to the interrupt. The lesser the latency faster is the response to the interrupt event.



9. How will you use a variable defined in source file1 inside source file2?

We can achieve this by making use of the “extern” keyboard. It allows the variable to be accessible from one file to another. This can be handled more cleanly by creating a header file that just consists of extern variable declarations. This header file is then included in the source files which uses the extern variables. Consider an example where we have a header file named `variables.h` and a source file named `sc_file.c` .

```
/* variables.h*/
extern int global_variable_x;
```

```
/* sc_file.c*/
#include "variables.h"  /* Header variables included */
#include <stdio.h>
void demoFunction(void)
{
   printf("Value of Global Variable X: %d\n", global_variable_x++);
}
```

10. What do you understand by segmentation fault?

A segmentation fault occurs most commonly and often leads to crashes in the programs. It occurs when a program instruction tries to access a memory address that is prohibited from getting accessed.

11. What are the differences between Inline and Macro Function?

Category	Macro Function	Inline Function
Compile-time expansion	Macro functions are expanded by the preprocessor at the compile time.	Inline functions are expanded by the compiler.
Argument Evaluation	Expressions passed to the Macro functions might get evaluated more than once.	Expressions passed to Inline functions get evaluated once.
Parameter Checking	Macro functions do not follow strict parameter data type checking.	Inline functions follow strict data type checking of the parameters.
Ease of debugging	Macro functions are hard to debug because it is replaced by the pre-processor as a textual representation which is not visible in the source code.	Easier to debug inline functions which is why it is recommended to be used over macro functions.

12. Is it possible for a variable to be both volatile and const?

The const keyword is used when we want to ensure that the variable value should not be changed. However, the value can still be changed due to external interrupts or events. So, we can use const with volatile keywords and it won't cause any problem.

13. Is it possible to declare a static variable in a header file?

Variables defined with static are initialized once and persists until the end of the program and are local only to the block it is defined. A static variables declaration requires definition. It can be defined in a header file. But if we do so, a private copy of the variable of the header file will be present in each source file the header is included. This is not preferred and hence it is not recommended to use static variables in a header file.

14. What do you understand by the pre-decrement and post-decrement operators?

The Pre-decrement operator (`--operand`) is used for decrementing the value of the variable by 1 before assigning the variable value.

```
#include < stdio.h >
int main(){
    int x = 100, y;

    y = --x;   //pre-decrememt operators  -- first decrements the value and then it is

    printf("y = %d\n", y);  // Prints 99
    printf("x = %d\n", x);  // Prints 99
    return 0;
}
```

The Post-decrement operator (`operand--`) is used for decrementing the value of a variable by 1 after assigning the variable value.

```
#include < stdio.h >
int main(){
    int x = 100, y;

    y = x--;   //post-decrememt operators  -- first assigns the value and then it is de

    printf("y = %d\n", y);  // Prints 100
    printf("x = %d\n", x);  // Prints 99
    return 0;
}
```

15. What is a reentrant function?

A function is called reentrant if the function can be interrupted in the middle of the execution and be safely called again (re-entered) to complete the execution. The interruption can be in the form of external events or signals or internal signals like call or jump. The reentrant function resumes at the point where the execution was left off and proceeds to completion.

Embedded C Interview Questions for Experienced

16. What kind of loop is better - Count up from zero or Count Down to zero?

Loops that involve count down to zero are better than count-up loops. This is because the compiler can optimize the comparison to zero at the time of loop termination. The processors need not have to load both the loop variable and the maximum value for comparison due to the optimization. Hence, count down to 0 loops are always better.

17. What do you understand by a null pointer in Embedded C?

A null pointer is a pointer that does not point to any valid memory location. It is defined to ensure that the pointer should not be used to modify anything as it is invalid. If no address is assigned to the pointer, it is set to `NULL` .

Syntax:

```
data_type *pointer_name = NULL;
```

One of the uses of the null pointer is that once the memory allocated to a pointer is freed up, we will be using NULL to assign to the pointer so that it does not point to any garbage locations.

18. Following are some incomplete declarations, what do each of them mean?

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

- The first two declaration points 1 and 2 mean the same. It means that the variable `x` is a read-only constant integer.
- The third declaration represents that the variable `a` is a pointer to a constant integer. The integer value can't be modified but the pointer can be modified to point to other locations.
- The fourth declaration means that the variable `x` is a constant pointer to an integer value. It means that the integer value can be changed, but the pointer can't be made to point to anything else.
- The last declaration means that the variable x is a constant pointer to a constant integer which means that neither the pointer can point to a different location nor the integer value can be modified.

19. Why is the statement ++i faster than i+1?

- ++i instruction uses single machine instruction like INR (Increment Register) to perform the increment.
- For the instruction i+1, it requires to load the value of the variable i and then perform the INR operation on it. Due to the additional load, ++i is faster than the i+1 instruction.

20. What are the reasons for segmentation fault in Embedded C? How do you avoid these errors?

Following are the reasons for the segmentation fault to occur:

- While trying to dereference NULL pointers.
- While trying to write or update the read-only memory or non-existent memory not accessible by the program such as code segment, kernel structures, etc.
- While trying to dereference an uninitialized pointer that might have been pointing to invalid memory.
- While trying to dereference a pointer that was recently freed using the free function.
- While accessing the array beyond the boundary.

Some of the ways where we can avoid Segmentation fault are:

- **Initializing Pointer Properly:** Assign addresses to the pointers properly. For instance:
  - We can also assign the address of the matrix, vectors or using functions like calloc, malloc etc.
  - Only important thing is to assign value to the pointer before accessing it.

```
int varName;
int *p = &varName;
```

- **Minimize using pointers:** Most of the functions in Embedded C such as scanf, require that address should be sent as a parameter to them. In cases like these, as best practices, we declare a variable and send the address of that variable to that function as shown below:

```
int x;
scanf("%d",&x);
```

In the same way, while sending the address of variables to custom-defined functions, we can use the & parameter instead of using pointer variables to access the address.

```
int x = 1;
x = customFunction(&x);
```

- **Troubleshooting:** Make sure that every component of the program like pointers, array subscripts, & operator, * operator, array accessing, etc as they can be likely candidates for segmentation error. Debug the statements line by line to identify the line that causes the error and investigate them.

21. Is it recommended to use printf() inside ISR?

printf() is a non-reentrant and thread-safe function which is why it is not recommended to call inside the ISR.

22. Is it possible to pass a parameter to ISR or return a value from it?

An ISR by nature does not allow anything to pass nor does it return anything. This is because ISR is a routine called whenever hardware or software events occur and is not in control of the code.

23. What Is a Virtual Memory in Embedded C and how can it be implemented?

Virtual memory is a means of allocating memory to the processes if there is a shortage of physical memory by using an automatic allocation of storage. The main advantage of using virtual memory is that it is possible to have larger virtual memory than physical memory. It can be implemented by using the technique of paging.

Paging works as follows:

- Whenever a process needs to be executed, it would be swapped into the memory by using a lazy swapper called a pager.
- The pager tries to guess which page needs to get access to the memory based on a predefined algorithm and swaps that process. This ensures that the whole process is not swapped into the memory, but only the necessary parts of the process are swapped utilizing pages.
- This decreases the time taken to swap and unnecessary reading of memory pages and reduces the physical memory required.

24. What is the issue with the following piece of code?

```
int square (volatile int *p){
    return (*p) * (*p) ;
}
```

From the code given, it appears that the function intends to return the square of the values pointed by the pointer p. But, since we have the pointer point to a volatile integer, the compiler generates code as below:

```
int square ( volatile int *p){
    int x , y;
    x = *p ;
    y = *p ;
    return x * y ;
}
```

Since the pointer can be changed to point to other locations, it might be possible that the values of the x and y would be different which might not even result in the square of the numbers. Hence, the correct way for achieving the square of the number is by coding as below:

```
long square (volatile int *p ){
    int x ;
    x = *p ;
    return x*x;
}
```

25. The following piece of code uses __interrupt keyword to define an ISR. Comment on the correctness of the code.

```
__interrupt double calculate_circle_area (double radius){
    double circle_area  = PI * radius * radius;
    printf ( 'Area = %f ' , circle_area);
    return circle_area;
}
```

Following things are wrong with the given piece of code:

- ISRs are not supposed to return any value. The given code returns a value of datatype double.
- It is not possible to pass parameters to ISRs. Here, we are passing a parameter to the ISR which is wrong.
- It is not advisable to have printf inside the ISR as they are non-reentrant and thereby it impacts the performance.

26. What is the result of the below code?

```
void demo(void){
    unsigned int x = 10 ;
    int y = -40;
    if(x+y > 10) {
        printf("Greater than 10");
    } else {
        printf("Less than or equals 10");
    }
}
```

In Embedded C, we need to know a fact that when expressions are having signed and unsigned operand types, then every operand will be promoted to an unsigned type. Herem the -40 will be promoted to unsigned type thereby making it a very large value when compared to 10. Hence, we will get the statement “Greater than 10” printed on the console.

27. What are the reasons for Interrupt Latency and how to reduce it?

Following are the various causes of Interrupt Latency:

- **Hardware**: Whenever an interrupt occurs, the signal has to be synchronized with the CPU clock cycles. Depending on the hardware of the processor and the logic of synchronization, it can take up to 3 CPU cycles before the interrupt signal has reached the processor for processing.
- **Pipeline**: Most of the modern CPUs have instructions pipelined. Execution happens when the instruction has reached the last stage of the pipeline. Once the execution of an instruction is done, it would require some extra CPU cycles to refill the pipeline with instructions. This contributes to the latency.

Interrupt latency can be reduced by ensuring that the ISR routines are short. When a lower priority interrupt gets triggered while a higher priority interrupt is getting executed, then the lower priority interrupt would get delayed resulting in increased latency. In such cases, having smaller ISR routines for lower priority interrupts would help to reduce the delay.

Also, better scheduling and synchronization algorithms in the processor CPU would help minimize the ISR latency.

28. Is it possible to protect a character pointer from accidentally pointing it to a different address?

It can be done by defining it as a constant character pointer. const protects it from modifications.

29. What do you understand by Wild Pointer? How is it different from Dangling Pointer?

A pointer is said to be a wild pointer if it has not been initialized to NULL or a valid memory address. Consider the following declaration:

```
int *ptr;
*ptr = 20;
```

Here the pointer ptr is not initialized and in the next step, we are trying to assign a valid value to it. If the ptr has a garbage location address, then that would corrupt the upcoming instructions too.

If we are trying to de-allocate this pointer and free it as well using the free function, and again if we are not assigning the pointer as NULL or any valid address, then again chances are that the pointer would still be pointing to the garbage location and accessing from that would lead to errors. These pointers are called dangling pointers.

30. What are the differences between the following 2 statements #include "..." and #include <...>?

Both declarations specify for the files to be included in the current source file. The difference is in how and where the preprocessor looks for including the files. For `#include "..."` , the preprocessor just searches for the file in the current directory as where the source file is present and if not found, it proceeds to search in the standard directories specified by the compiler. Whereas for the `#include <...>` declaration, the preprocessor looks for the files in the compiler designated directories where the standard library files usually reside.

31. When does a memory leak occur? What are the ways of avoiding it?

Memory leak is a phenomenon that occurs when the developers create objects or make use of memory to help memory and then forget to free the memory before the completion of the program. This results in reduced system performance due to the reduced memory availability and if this continues, at one point, the application can crash. These are serious issues for applications involving servers, daemons, etc that should ideally never terminate.

Example of Memory Leak:

```
#include <stdlib.h>

void memLeakDemo()
{
   int *p = (int *) malloc(sizeof(int));

   /* Some set of statements */

   return; /* Return from the function without freeing the pointer p*/
}
```

In this example, we have created pointer p inside the function and we have not freed the pointer before the completion of the function. This causes pointer p to remain in the memory. Imagine 100s of pointers like these. The memory will be occupied unnecessarily and hence resulting in a memory leak.

We can avoid memory leaks by always freeing the objects and pointers when no longer required. The above example can be modified as:

```
#include <stdlib.h>;

void memLeakFix()
{
   int *p = (int *) malloc(sizeof(int));

   /* Some set of statements */

   free(p); // Free method to free the memory allocated to the pointer p
   return;
}
```

Embedded C Programming

32. Write an Embedded C program to multiply any number by 9 in the fastest manner.

This can be achieved by involving bit manipulation techniques - Shift left operator as shown below:

```
#include<stdio.h>
void main(){
    int num;
    printf("Enter number: ");
    scanf("%d",&num);
    printf("%d", (num<<3)+num);
}
```

33. How will you swap two variables? Write different approaches for the same.

- **Using Extra Memory Space:**

```
int num1=20, num2=30, temp;
temp = num1;
num1 = num2;
num2 = temp;
```

- **Using Arithmetic Operators:**

```
int num1=20, num2=30;
num1=num1 + num2;
num2=num1 - num2;
num1=num1 - num2;
```

- **Using Bit-Wise Operators:**

```
int num1=20, num2=30;
num1=num1 ^ num2;
num2=num2 ^ num1;
num1=num1 ^ num2;
```

- **Using One-liner Bit-wise Operators:**

```
int num1=20, num2=30;
num1^=num2^=num1^=num2;
```

The order of evaluation here is right to left.

- **Using One-liner Arithmetic Operators:**

```
int num1=20, num2=30;
num1 = (num1+num2)-(num2=num1);
```

Here the order of evaluation is from left to right.

34. Write a program to check if a number is a power of 2 or not.

We can do this by using bitwise operators.

```
void main (){
    int num;
    printf ("Enter any no:");
    scanf ("%d", &num);
    if (num & & ((num & num-1) == 0))
        printf ("Number is a power of 2");
    else
        printf ("Number is not a power of 2");
}
```

35. Write a program to print numbers from 1 to 100 without making use of conditional operators?

```
void main (){
   int i=0;
   while (100 – i++)
   printf ("%d", i);
}
```

36. Write a MIN macro program that takes two arguments and returns the smallest of both arguments.

```
#define MIN(NUM1,NUM2) ( (NUM1) <= (NUM2) ? (NUM1) : (NUM2) )
```

Links to More Interview Questions

C Interview Questions	Php Interview Questions	C Sharp Interview Questions
Web Api Interview Questions	Hibernate Interview Questions	Node Js Interview Questions
Cpp Interview Questions	Oops Interview Questions	Devops Interview Questions
Machine Learning Interview Questions	Docker Interview Questions	Mysql Interview Questions
Css Interview Questions	Laravel Interview Questions	Asp Net Interview Questions
Django Interview Questions	Dot Net Interview Questions	Kubernetes Interview Questions
Operating System Interview Questions	React Native Interview Questions	Aws Interview Questions
Git Interview Questions	Java 8 Interview Questions	Mongodb Interview Questions
Dbms Interview Questions	Spring Boot Interview Questions	Power Bi Interview Questions
Pl Sql Interview Questions	Tableau Interview Questions	Linux Interview Questions
Ansible Interview Questions	Java Interview Questions	Jenkins Interview Questions