

100 embedded c interview questions, your interviewer might ask



100 EMBEDDED C INTERVIEW QUESTIONS

In my previous post, I have created a collection of “c interview questions” that is liked by many people. I have also got the response to create a list of interview questions on “embedded c”. So here I have tried to create some collection of embedded c interview questions that might be asked by your interviewer.

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 (<https://aticleworld.com/understanding-volatile-qualifier-in-c/>), 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 memory is the only way to check the unpredictable change of the value.

What is the use of volatile keyword?

The volatile keyword is mainly used where we directly deal with GPIO, interrupt or I/O Register. It is also used where a global variable or buffer is shared between the threads.

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

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

e.g,

```
const int a = 0;
```

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

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

```
volatile int a;
```

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

Can a variable be both constant and volatile in C?

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

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

Syntax of declaration,

```
int volatile * const PortRegister;
```

How to read the above declaration,

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



Can we have a volatile pointer?

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

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

The Proper place to use the volatile keyword?

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;

uint32_t GetRecvData()
{
    //Code to recv data
    while (((*pStatusReg) & COM_STATUS_BIT) == 0)
    {
        // Wait until 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;
}
```

What is ISR?

An ISR refers to the Interrupt Service Routines. These are procedures stored at specific memory addresses which are called when a certain type of interrupt occurs. The Cortex-M processors family has the NVIC that manage the execution of the interrupt.

Can we pass any parameter and return a value from the ISR?

An ISR returns nothing and not allow to pass any parameter. An ISR is called when a hardware or software event occurs, it is not called by the code, so that's the reason no parameters are passed into an ISR.

In the above line, we have already read that the ISR is not called by the code, so there is no calling code to read the returned values of the ISR. It is the reason that an ISR is not returned any value.

What is interrupt latency?

It is an important question that is asked by the interviewer to test the understanding of Interrupt. Basically, interrupt latency is the number of clock cycles that is taken by the processor to respond to an interrupt request. These number of the clock cycle is count between the assertions of the interrupt request and first instruction of the interrupt handler.

Interrupt Latency on the Cortex-M processor family

The Cortex-M processors have the very low interrupt latency. In below table, I have mentioned, Interrupt latency of Cortex- M processors with zero wait state memory systems.

Processors	Cycles with zero wait state memory
Cortex-M0	16
Cortex-M0+	15
Cortex-M3	12
Cortex-M4	12
Cortex-M7	12

How do you measure interrupt latency?

With the help of the oscilloscope, we can measure the interrupt latency. You need to take following steps.

- & First takes two GPIOs.
- & Configure one GPIO to generate the interrupt and second for the toggling (if you want you can attach an LED).
- & Monitor the PIN (using the oscilloscope or analyzer) which you have configured to generate the interrupt.
- & Also, monitor (using the oscilloscope or analyzer) the second pin which is toggled at the beginning of the interrupt service routine.
- & When you will generate the interrupt then the signal of the both GPIOs will change.

The interval between the two signals (interrupt latency) may be easily read from the instrument.

How to reduce the interrupt latency?

The interrupt latency depends on many factors, some factor I am mentioning in below statements.

- & Platform and interrupt controller.

- & CPU clock speed.
- & Timer frequency
- & Cache configuration.
- & Application program.

So using the proper selection of platform and processor we can easily reduce the interrupt latency. We can also reduce the interrupt latency by making the ISR shorter and avoid to calling a function within the ISR.

What are the causes of Interrupt Latency?

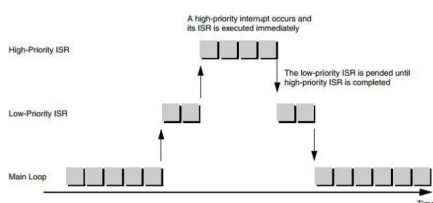
- & The first delay is typically caused by hardware: The interrupt request signal needs to be synchronized to the CPU clock. Depending on the synchronization logic, up to 3 CPU cycles may expire before the interrupt request has reached the CPU core.
- & The CPU will typically complete the current instruction, which may take several cycles. On most systems, divide, push- multiple or memory-copy instructions are the most time-consuming instructions to execute. On top of the cycles required by the CPU, additional cycles are often required for memory accesses. In an ARM7 system, the instruction STMDB SP!, {R0-R11, LR} typically is the worst case instruction, storing 13 registers of 32-bits each to the stack, and takes 15 clock cycles to complete.
- & The memory system may require additional cycles for wait states.
- & After completion of the current instruction, the CPU performs a mode switch or pushes registers on the stack (typically PC and flag registers). Modern CPUs such as ARM generally perform a mode switch, which takes fewer CPU cycles than saving registers.
- & Pipeline fill: Most modern CPUs are pipelined. Execution of an instruction happens in various stages of the pipeline. An instruction is executed when it has reached its final stage of the pipeline. Since the mode switch has flushed the pipeline, a few extra cycles are required to refill the pipeline.

What is nested interrupt?

In a nested interrupt system, an interrupt is allowed to any time and anywhere even an ISR is being executed. But, only the highest priority ISR will be executed immediately. The second highest priority ISR will be executed after the highest one is completed.

The rules of a nested interrupt system are:

- & All interrupts must be prioritized.
- & After initialization, any interrupts are allowed to occur anytime and anywhere.
- & If a low-priority ISR is interrupted by a high-priority interrupt, the high-priority ISR is executed.
- & If a high-priority ISR is interrupted by a low-priority interrupt, the high-priority ISR continues executing.
- & The same priority ISRs must be executed by time order



If you want to learn STM32 from scratch, you should follow this course “[Mastering Microcontroller with Embedded Driver Development](https://click.linksynergy.com/deeplink?id=UajS5soDmWY&mid=39197&murl=https%3A%2F%2Fwww.udemy.com%2Fmastering-microcontroller-with-peripheral-driver-development%2F) (https://click.linksynergy.com/deeplink?id=UajS5soDmWY&mid=39197&murl=https%3A%2F%2Fwww.udemy.com%2Fmastering-microcontroller-with-peripheral-driver-development%2F)“. The course contains video lectures of 18.5-hours length covering all topics like, Microcontroller & Peripheral Driver Development for STM32 GPIO, I2C, SPI, USART using Embedded C.



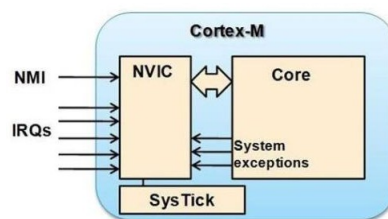
(https://click.linksynergy.com/deeplink?id=UajS5soDmWY&mid=39197&murl=https%3A%2F%2Fwww.udemy.com%2Fmastering-microcontroller-with-peripheral-driver-development%2F)

[Enroll In Course \(https://click.linksynergy.com/deeplink?id=UajS5soDmWY&mid=39197&murl=https%3A%2F%2Fwww.udemy.com%2Fmastering-microcontroller-with-peripheral-driver-development%2F\)](https://click.linksynergy.com/deeplink?id=UajS5soDmWY&mid=39197&murl=https%3A%2F%2Fwww.udemy.com%2Fmastering-microcontroller-with-peripheral-driver-development%2F)

What is NVIC in ARM Cortex?

The Nested Vector Interrupt Controller (NVIC) in the Cortex-M processor family is an example of an interrupt controller with extremely flexible interrupt priority management. It enables programmable priority levels, automatic nested interrupt support, along with support for multiple interrupt masking, whilst still being very easy to use by the programmer.

The Cortex-M3 and Cortex-M4 processors the NVIC supports up to 240 interrupt inputs, with 8 up to 256 programmable priority levels



Can we use any function inside ISR?

Yes, you can call a function within the ISR but it is not recommended because it can increase the interrupt latency and decrease the performance of the system. If you want to call a nested function within the ISR, you need to read the datasheet of your microcontroller because some vendors have a limit to how many calls can be nested.

One important point also needs to remember that function which is called from the ISR should be re-entrant. If the called function is not re-entrant, it could create the issues.

For example,

If the function is not reentrant and supposes that it is called by another part of the code beside the ISR. So the problem will be invoked when if the ISR calls the same function which is already invoked outside of the ISR?

Can we change the interrupt priority level of Cortex-M processor family?


Yes, we can.

What is the start-up code?


A start-up code is called prior to the main function, it creates a basic platform for the application. It is a small block of code that is written in assembly language.

There are following parts of the start-up code.

- & Declaration of the Stack area.
- & Declaration of the Heap area.
- & Vector table.
- & Reset handler code.
- & Other exception handler code.

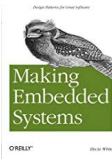


C Programming Language, 2nd Edition (https://aax-us-east.amazon-adsystem.com/x/c/QhI8hDUxdOncJ6OQu6mi-YkAAAFrZBID6AEAAAFKAZ9lcWc/https://www.amazon.com/Programming-Language-2nd-Edition-Kernighan/dp/0611351103628/ref=sm_n_se_dkp_VN_pr_sea_0_0?adId=0131103628&creativeASIN=0131103628&linkId=0e96fb5178b3c0d08d9...)




Embedded C Programming: Techniques and Application... (https://aax-us-east.amazon-adsystem.com/x/c/QhI8hDUxdOncJ6OQu6mi-YkAAAFrZBID6AEAAAFKAZ9lcWc/https://www.amazon.com/Embedded-C-Programming-Techniques-and-Applications-of-C-and-PIC-MCU-Mark-Siegemund/dp/012801314113141/ref=sm_n_se_dkp_VN_pr_sea_1_0?adId=0128013141&creativeASIN=0128013141&linkId=0e96fb5178b3c0d08d9...)

Ads by Amazon (https://aax-us-east.amazon-adsystem.com/x/c/QhI8hDUxdOncJ6OQu6mi-YkAAAFrZBID6AEAAAFKAZ9lcWc/https://affiliate-program.amazon.com/home/ads/ref=sm_n_se_dkp_VN_logo?adId=logo&creativeASIN=logo&linkId=0e96fb5178b3c0d08d9383a937f63edd&tag=articleworld02-20&linkCode=w42&ref-refURL=https%3A%2F%2Farticleworld.com%2Fembedded-c-interview-questions-2%2F&slotNum=1&imprToken=PacoyqB5K.-KajXbujSXGg&adType=smart&adFormat=grid&impressionTimestamp=1562496864063&ac-ms-src=nsa-ads&cid=nsa-ads)
(https://aax-us-east.amazon-adsystem.com/x/c/QhI8hDUxdOncJ6OQu6mi-YkAAAFrZBID6AEAAAFKAZ9lcWc/https://www.amazon.com/adprefs/ref=sm_n_se_dkp_VN_ac?tag=articleworld02-20&linkCode=w42)



Making Embedded Systems: Design Patterns for Grea... (https://aax-us-east.amazon-adsystem.com/x/c/QhI8hDUxdOncJ6OQu6mi-YkAAAFrZBID6AEAAAFKAZ9lcWc/https://www.amazon.com/Making-Embedded-Systems-Design-Patterns-for-Green-Embedded-Systems-Peter-Barton/dp/146449302149/ref=sm_n_se_dkp_VN_pr_sea_0_1?adId=1449302149&creativeASIN=1449302149&linkId=0e96fb5178b3c0d08d9...)



Test Driven Development for Embedded C (Pragmat... (https://aax-us-east.amazon-adsystem.com/x/c/QhI8hDUxdOncJ6OQu6mi-YkAAAFrZBID6AEAAAFKAZ9lcWc/https://www.amazon.com/Driven-Programmers/dp/193435662X/ref=sm_n_se_dkp_VN_pr_sea_1_1?adId=193435662X&creativeASIN=193435662X&linkId=0e96fb5178b3c0d08d9...)

What are the start-up code steps?

Start-up code for C programs usually consists of the following actions, performed in the order described:

- & Disable all interrupts.

- & Copy any initialized data from ROM to RAM.
 - & Zero the uninitialized data area.
 - & Allocate space for and initialize the stack.
 - & Initialize the processor's stack pointer.
 - & Create and initialize the heap.
 - & Enable interrupts.
 - & Call main.
-

Infinite loops often arise in embedded systems. How do you code an infinite loop in C?

In embedded systems, infinite loops are generally used. If I talked about a small program to control a led through the switch, in that scenario an infinite loop will be required if we are not going through the interrupt.

There are the different way to create an infinite loop, here I am mentioning some methods.

Method 1:

```
while(1)
{
    // task
}
```

Method 2:

```
for(;;)
{
    // task
}
```

Method 3:

```
Loop:
goto Loop;
```

How to access the fixed memory location in embedded C?

It is a very basic question that is generally asked by the interviewer.

Let's take an example:

Suppose in an application, you have required accessing a fixed memory address.

```
//Memory address, you want to access
#define RW_FLAG 0x1FFF7800

//Pointer to access the Memory address volatile
uint32_t *GagAddress = NULL;
```



```
//variable to stored the read value
uint32_t readData = 0;

//Assign address to the pointer
GagAddress = (volatile uint32_t *)RW_FLAG;

//Read value from memory
* GagAddress = 12; // Write

//Write value to the memory
readData = * GagAddress;
```

Difference between RISC and CISC processor?

The RISC (reduced instruction set computer) and CISC (Complex instruction set computer) are the processors ISA (instruction set architecture).

There are following difference between both architecture:

	RISC	CISC
Acronym	It stands for 'Reduced Instruction Set Computer'.	It stands for 'Complex Instruction Set Computer'.
Definition	The RISC processors have a smaller set of instructions with few addressing nodes.	The CISC processors have a larger set of instructions with many addressing nodes.
Memory unit	It has no memory unit and uses a separate hardware to implement instructions.	It has a memory unit to implement complex instructions.
Program	It has a hard-wired unit of programming.	It has a micro-programming unit.
Design	It is a complex compiler design.	It is an easy compiler design.
Calculations	The calculations are faster and precise.	The calculations are slow and precise.
Decoding	Decoding of instructions is simple.	Decoding of instructions is complex.
Time	Execution time is very less.	Execution time is very high.
External memory	It does not require external memory for calculations.	It requires external memory for calculations.
Pipelining	Pipelining does function correctly.	Pipelining does not function correctly.
Stalling	Stalling is mostly reduced in processors.	The processors often stall.
Code expansion	Code expansion can be a problem.	Code expansion is not a problem.
Disc space	The space is saved.	The space is wasted.
Applications	Used in high-end applications such as video processing, telecommunications and image processing.	Used in low-end applications such as security systems, home automations, etc.

Images Courtesy: ics.uci.edu

What is the stack overflow?

If your program tries to access the beyond the limit of the available stack memory then stack overGow occurs. In other word you can say that a stack overGow occurs if the call stack pointer exceeds the stack boundary.

If stack overGow occurs, the program can crash or you can say that segmentation fault that is the result of the stack overGow.

What is the cause of the stack overflow?

In the embedded application we have a little amount of stack memory as compare to the desktop application. So we have to work on embedded application very carefully either we can face the stack overGow issues that can be a cause of the application crash.

Here, I have mentioned some causes of unwanted use of the stack.

- & Improper use of the recursive function.
 - & Passing to much arguments in the function.
 - & Passing a structure directly into a function.
 - & Nested function calls.
 - & Creating a huge size local array.
-

What is the difference between I2c and SPI communication Protocol?

In the embedded system, I2C and SPI both play an important role. Both communication protocols are the example of the synchronous communication but still, both have some important difference.

The important difference between the I2C and SPI communication protocol.

- & I2C support half duplex while SPI is the full duplex communication.
- & I2C requires only two wire for communication while SPI requires three or four wire for communication (depends on requirement).
- & I2C is slower as compared to the SPI communication.
- & I2C draws more power than SPI.
- & I2C is less susceptible to noise than SPI.
- & I2C is cheaper to implement than the SPI communication protocol.
- & I2C work on wire and logic and it has a pull-up resistor while there is no requirement of pull-up resistor in case of the SPI.
- & In I2C communication we get the acknowledgment bit after each byte, it is not supported by the SPI communication protocol.
- & I2C ensures that data sent is received by the slave device while SPI does not verify that data is received correctly.
- & I2C support the multi-master communication while multi-master communication is not supported by the SPI.
- & One great difference between I2C and SPI is that I2C supports multiple devices on the same bus without any additional select lines (work on the basis of device address) while SPI requires additional signal (slave select lines) lines to manage multiple devices on the same bus.
- & I2C supports arbitration while SPI does not support the arbitration.
- & I2C support the clock stretching while SPI does not support the clock stretching.
- & I2C can be locked up by one device that fails to release the communication bus.
- & I2C has some extra overhead due to start and stop bits.

- & I2C is better for long distance while SPI is better for the short distance.
- & In the last I2C developed by NXP while SPI by Motorola.

What is the difference between Asynchronous and Synchronous Communication?

There are following difference between the asynchronous and synchronous communication.

Asynchronous Communication	Synchronous Communication
There is no common clock signal between the sender and receivers.	Communication is done by a shared clock.
Sends 1 byte or character at a time.	Sends data in the form of blocks or frames.
Slow as compare to synchronous communication.	Fast as compare to asynchronous
communication. Overhead due to start and stop bit.	Less overhead.
Ability to communicate long distance.	Less as compared to asynchronous
communication. A start and stop bit used for the data synchronization.	A shared clock is used for the data
synchronization.	
Economical	Costly
RS232, RS485	I2C, SPI.

What is the difference between RS232 and RS485?

The RS232 and RS485 is an old serial interface. Both serial interfaces are the standard for the data communication. This question is also very important and generally ask by an interviewer.

Some important difference between the RS232 and RS485

Parameter	RS232	RS485
Line configuration	Single –ended	differential
Numbers of devices	1 transmitter 1 receiver	32 transmitters 32 receivers
Mode of operation	Simplex or full duplex	Simplex or half duplex
Maximum cable length	50 feet	4000 feet
Maximum data rate	20 Kbits/s	10 Mbits/s
signaling	unbalanced	balanced
Typical logic levels	+5 ~ +-15V	+1.5 ~ +-6V
Minimum receiver input impedance	3 ~ 7 K-ohm	12 K-ohm
Receiver sensitivity	+3V	+200mV

What is the difference between Bit Rate and Baud Rate?

Bit Rate	Baud Rate
Bit rate is the number of bits per second.	Baud rate is the number of signal units per second.
It determines the number of bits traveled per second.	It determines how many times the state of a signal is changing.
Cannot determine the bandwidth.	It can determine how much bandwidth is required to send the signal.
This term generally used to describe the processor efficiency.	This term generally used to describe the data transmission over the channel.
Bit rate = baud rate x the number of bits per signal unit	Baud rate = bit rate / the number of bits per signal unit

What is segmentation fault in C?

A segmentation fault is a common problem that causes programs to crash. A core file (core dumped file) also associated with segmentation fault that is used by the developer to finding the root cause of the crashing (segmentation fault).

Generally, the segmentation fault occurs when a program tried to access a memory location that it is not allowed to access or tried to access a memory location in a way that is not allowed (tried to access read-only memory).

What are the common causes of segmentation fault in C?

There are many reasons for the segmentation fault, here I am listing some common causes of the segmentation fault.

- & Dereferencing NULL pointers.
- & Tried to write read-only memory (such as code segment).
- & Trying to access a nonexistent memory address (outside process's address space).
- & Trying to access memory the program does not have rights to (such as kernel structures in process context).
- & Sometimes dereferencing or assigning to an uninitialized pointer (because might point an invalid memory) can be the cause of the segmentation fault.
- & Dereferencing the freed memory (after calling the free function) can also be caused by the segmentation fault.
- & A stack overflow is also caused by the segmentation fault.
- & A buffer overflow (try to access the array beyond the boundary) is also cause of the segmentation fault.

What is the difference between Segmentation fault and Bus error?

In case of segmentation fault, SIGSEGV (11) signal is generated. Generally, a segmentation fault occurs when the program tries to access the memory to which it doesn't have access to.

In below I have mentioned some scenarios where SIGSEGV signal is generated.

- & When trying to de-referencing a NULL pointer.
- & Trying to access memory which is already de-allocated (trying to use dangling pointers).
- & Using uninitialized pointer(wild pointer).
- & Trying to access memory that the program doesn't own (eg. trying to access an array element out of array bounds).

In case of a BUS error, SIGBUS (10) signal is generated. The Bus error issue occurs when a program tries to access an invalid memory or unaligned memory. The bus error comes rarely as compared to the segmentation fault.

In below I have mentioned some scenarios where SIGBUS signal is generated.

- & Non-existent address.
- & Unaligned access.
- & Paging errors

Size of the integer depends on what?

The C standard is explained that the minimum size of the integer should be 16 bits. Some programming 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 (`uint8_t`, `uint16_t`, `uint32_t` ..) in place of standard int.



Are integers signed or unsigned?

In standard C language, 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 \(https://aticleworld.com/signed-and-unsigned-integers/\)](https://aticleworld.com/signed-and-unsigned-integers/)

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

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

- & A signed integer can store the positive and negative value both but beside it unsigned integer can only store the positive value.
- & The range of nonnegative values of a signed integer type is a sub-range of the corresponding unsigned integer type.

For example,

Assuming size of the integer is 2 bytes.

signed int -32768 to +32767

unsigned int 0 to 65535

- & When computing the unsigned integer, it never gets overflow because if the computation result is greater than the largest value of the unsigned integer type, it is reduced modulo the number that is one greater than the largest value that can be represented by the resulting type.

For example,

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

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

What is the difference between a macro and a function?

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.

What is the difference between typedef & Macros?

typedef:

The C language provides a very important keyword typedef (<https://aticleworld.com/typedef-in-c/>) 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.

What do you mean by enumeration in C?

In C language `enum` (<https://aticleworld.com/seven-important-points-enum-c-language/>) is user-defined data type and it consists a set of named constant integer. Using the enum keyword, we can declare an enumeration type by using the enumeration tag (optional) and a list of named integer.

An enumeration increases the readability of the code and easy to debug in comparison of symbolic constant (macro).

The most important thing about the enum is that it follows the scope rule and compiler automatic assign the value to its member constant.

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

Syntax of enum,

```
enum Enumeration_Tag { Enumeration_List };
```

The Enumeration_Tag specifies the enumeration type name.

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

Example,

```
enum FLASH_ERROR { DEFRAGMENT_ERROR, BUS_ERROR};
```

What is the difference between const and macro?

- & The const keyword is handled by the compiler, in another hand, a macro is handled by the preprocessor directive.
 - & const is a qualifier that is modified the behavior of the identifier but macro is preprocessor directive.
 - & There is type checking is occurred with const keyword but does not occur with #define.
 - & const is scoped by C block, #define applies to a file.
 - & 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.
-

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)

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

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

Method 1((Using Arithmetic Operators):

```
#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;
}
```


What is meant by structure padding?

In the case of structure or union, the compiler inserts some extra bytes between the members of structure or union for the alignment, these extra unused bytes are called padding bytes (<https://aticleworld.com/data-alignment-and-structure-padding-bytes/>) and this technique is called padding.

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

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

What is the endianness?

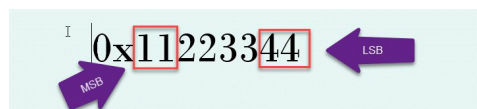
The endianness is the order of bytes to store data in memory and it also describes the order of byte transmission over a digital link. In memory data store in which order it depends on the endianness of the system, if the system is big-endian (<https://aticleworld.com/little-and-big-endian-importance/>) 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
<ul style="list-style-type: none">• Intel x86 and x86-64 series• Zilog Z80 (including Z180 and eZ80)• MOS Technology 6502	<ul style="list-style-type: none">• Motorola 68000 series• Xilinx Microblaze, SuperH,• IBM z/Architecture, Atmel AVR32.

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.

Write a c program 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 uCheckEndianness;
    uCheckEndianness.u32RawData = 0x11223344; //assign the value

    if (uCheckEndianness.au8DataBuff[0] == 0x44) //check the array first index value
    {
        printf("little-endian");
    }
    else if (uCheckEndianness.au8DataBuff[0] == 0x11) //check the array first index value
```

```
{  
printf("big-endian");
```

```
}  
  
return 0;  
}
```

How to convert little endian to big endian vice versa in C?

Method 1:

```
#include <stdio.h> #include  
<stdlib.h> #include  
<inttypes.h>  
  
//Function to change the endianness  
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;  
}
```

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](https://aticleworld.com/dynamic-memory-allocation-in-c/) 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.

What is the memory leak in C?

A memory leak (<https://aticleworld.com/problems-with-dynamic-memory-allocation/>) 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 then allocated memory does not allocate to another program or process until it gets free.

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.

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);`

```
#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

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

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.

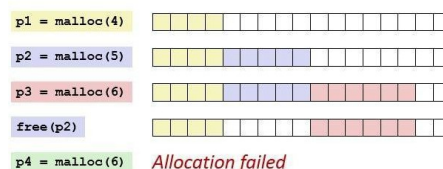
What is dynamic memory fragmentation?

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 (<https://aticleworld.com/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 program not able to use it. There are different types of free list allocation algorithms that used the free memory block efficiently.

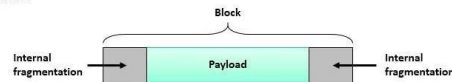
To understand the external fragmentation, consider a scenario where a program has 3 contiguous blocks of memory and the user frees the middle block of memory. In that scenario, you will not get a memory, if the required block of memory is larger than a single block of memory (but smaller or equal to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

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



For example,

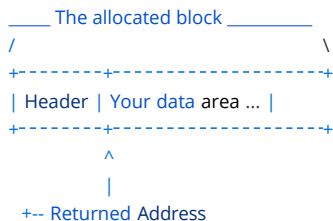
See the below code, the programmer may think that system will be allocated 8 * 100 (800) bytes of memory but due to bookkeeping (if 8 bytes) system will be allocated 8*100 extra bytes. This is an internal fragmentation, where 50% of the heap waste.

```
char *acBuffer[100]; int
main()
{
    int iLoop = 0;
    while(iLoop < 100)
    {
        acBuffer[iLoop] = malloc(8);
        ++iLoop;
    }
}
```

How is the free work in C?

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

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



For example,

```
#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;
}
```


What is a Function Pointer?

A [function pointer](https://aticleworld.com/use-of-function-pointer-in-c/) (<https://aticleworld.com/use-of-function-pointer-in-c/>) 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.

How to declare a pointer to a function in c?

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

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

For example,

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

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

e.g,

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

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

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

Where can the function pointers be used?

There are a lot of places, where the function pointers can be used. Generally, function pointers are used in the implementation of the callback function, [finite state machine](https://aticleworld.com/state-machine-using-c/) (<https://aticleworld.com/state-machine-using-c/>) and to provide the feature of [polymorphism in C](https://aticleworld.com/function-pointer-in-c-struct/) (<https://aticleworld.com/function-pointer-in-c-struct/>) language ...etc.

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

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

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

What is the output of the below code?

```
#include <stdio.h>

int main()
{
    int x = -15; x =

    x << 1;

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

Output:

undefined behavior.

What is the output of the below code?

```
#include <stdio.h> int

main()
{
    int x = -30; x =

    x >> 1;

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

Output:

implementation-defined.

Write a program to count set bits in an integer?

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

What is void or generic pointers in C?

A void pointer (<https://aticleworld.com/void-pointer-in-c/>) is a generic pointer. It has no associated data type that's why it can store the address of any type of object and type-casted to any types.

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

Syntax:

```
void * Pointer_Name;
```

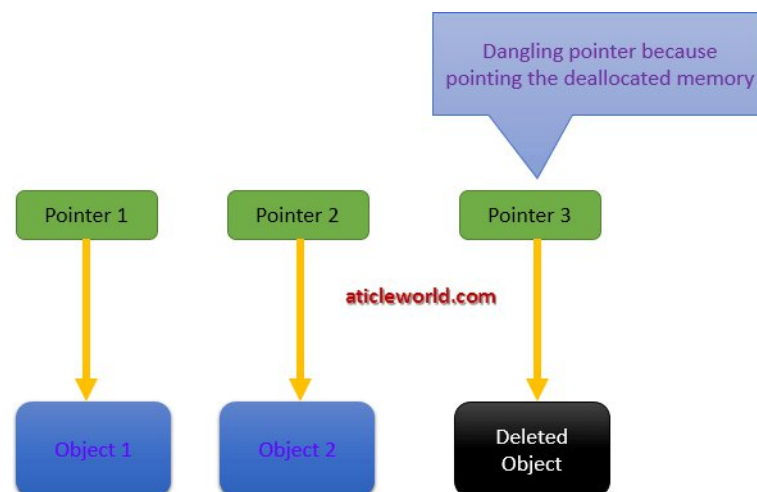
What is the advantage of a void pointer in C?

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. For more information see this link: [How to create generic Link List](#).

What are dangling pointers?

Generally, daggling pointers (<https://aticleworld.com/dangling-void-null-wild-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,




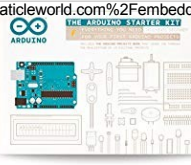
```
#include<stdio.h> #include<stdlib.h>

int main()
{
    int *piData = NULL;

    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.

 <p>ELEGOO UNO Project Super Starter Kit with Tu... (https://aax-us-east.amazon-adsystem.com/x/c/Qs0FJuPHnq4rfJyQozucEksAAAFrZBID1AEAAAFKAULY9...) Project-Tutorial-Controller- \$p3ro5je0c0ts/dp/B01D8KOZF4/ref=sm_n_se_dkp_VN_pr_sea_0_0? adId=B01D8K(O95Z8F)4&creativeASIN=B01D8KOZF4&linkId=49aef9aa83f7c2895... 20&linkCode=w42&ref-</p>	 <p>ARDUINO UNO R3 [A000066] (https://aax-us-east.amazon-adsystem.com/x/c/Qs0FJuPHnq4rfJyQozucEksAAAFrZBID1AEAAAFKAULY9...) A000066-ARDUINO-UNO- \$R230jd0p0/B008GRTSV6/ref=sm_n_se_dkp_VN_pr_sea_0_1? adId=B008GR(T7S32V)6&creativeASIN=B008GRTSV6&linkId=49aef9aa83f7c2895... 20&linkCode=w42&ref-</p>
 <p>Elegoo EL-KIT-008 Mega 2560 Project The Most C... (https://aax-us-east.amazon-adsystem.com/x/c/Qs0FJuPHnq4rfJyQozucEksAAAFrZBID1AEAAAFKAULY9...) \$K5T9-c09098-Project-Complete-Ultimate- Tutorial/dp/B0(15E4W4)NUUUA/ref=sm_n_se_dkp_VN_pr_sea_1_0? adId=B01EWNNUUUA&creativeASIN=B01EWNNUUUA&linkId=49aef9aa83f7c2...</p>	 <p>Arduino Starter Kit - English Official Kit With 170 Page... (https://aax-us-east.amazon-adsystem.com/x/c/Qs0FJuPHnq4rfJyQozucEksAAAFrZBID1AEAAAFKAULY9...) \$7600K4Engh- Official/dp/B00(796UgK)ZV0A/ref=sm_n_se_dkp_VN_pr_sea_1_1? adId=B009UKZV0A&creativeASIN=B009UKZV0A&linkId=49aef9aa83f7c2895...</p>

Ads by Amazon (https://aax-us-east.amazon-adsystem.com/x/c/Qs0FJuPHnq4rfJyQozucEksAAAFrZBID1AEAAAFKAULY9zo/https://affiliate-program.amazon.com/home/ads/ref=sm_n_se_dkp_VN_logo?adId=logo&creativeASIN=logo&linkId=49aef9aa83f7c2895f4870b99f96dd36&tag=aticleworld02-20&linkCode=w42&ref-refURL=https%3A%2F%2Faticleworld.com%2Fembedded-c-interview-questions-2%2F&slotNum=2&imprToken=NXPhkADQ7syylOaNyXKzfg&adType=smart&adMode=search&adFormat=grid&impressionTimestamp=1562496863492&ac-ms-src=nsa-ads&cid=nsa-ads)
(https://aax-us-east.amazon-adsystem.com/x/c/Qs0FJuPHnq4rfJyQozucEksAAAFrZBID1AEAAAFKAULY9zo/https://www.amazon.com/adprefs/ref=sm_n_se_dkp_VN_ac?tag=aticleworld02-20&linkCode=w42)

What is the wild pointer?

A pointer that is not initialized properly prior to its first use is known as the wild pointer (<https://aticleworld.com/dangling-void-null-wild-pointers/>). 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 (<https://aticleworld.com/pointers-in-c/>) that are not initialized either by the compiler or programmer begins as a wild pointer.

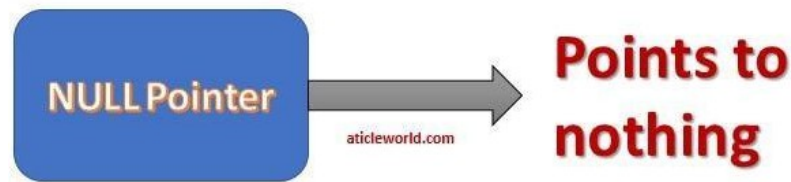
Note: Generally, compilers warn about the wild pointer.

Syntax,

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

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 (<https://aticleworld.com/dangling-void-null-wild-pointers/>) constant is converted to a pointer type, the resulting pointer, called a null pointer.



Syntax,

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

What are the post increment and decrement operators?

When we use 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 (<https://aticleworld.com/increment-decrement-operator-in-c/>) but the difference is that the value of the operand is decremented by 1.

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

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 Gash 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
}
```

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

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] = {100,200,300,400,500};

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

What does the keyword const mean?

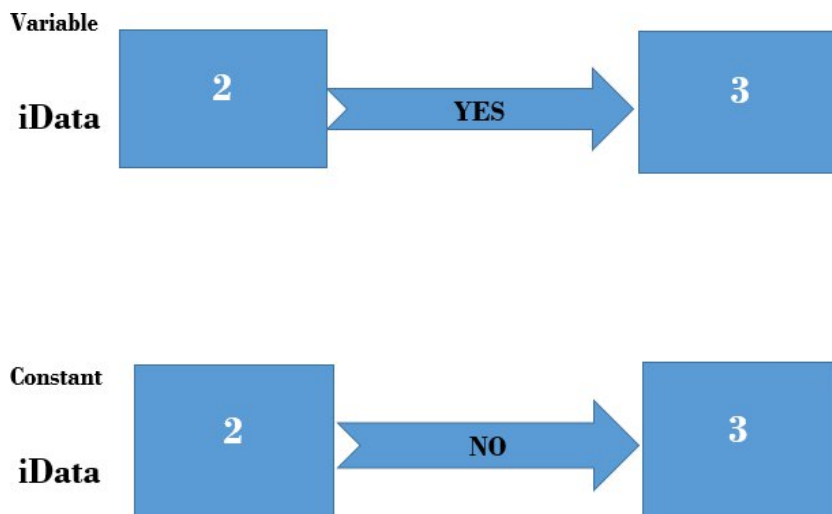
A const is only a qualifier, it changes the behavior of a variable and makes it read-only type. When we want to make an object read-only type, then we have to declare it as const.

Syntax

`const DataType Identifier = Value;`

e.g.

`const int iData = 0`



At the time of declaration, const qualifier (<https://aticleworld.com/const-qualifier-in-c-language/>) only gives the direction to the compiler that the value of declaring object could not be changed. In simple word, const means not modifiable (cannot assign any value to the object at the runtime).

When should we use const in a C program?

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

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

What is the meaning of below declarations?

1. `const int a;`
 2. `int const a;`
 3. `const int *a;`
 4. `int* const a;`
 5. `int const * a const;`
-
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 pointed integer and pointer both are not modifiable.
-

Differentiate between a constant pointer and pointer to a constant?

Constant pointer:

A constant pointer (<https://aticleworld.com/important-question-related-to-const/>) is a pointer whose value (pointed address) is not modifiable. If you will try to modify the pointer value, you will get the compiler error.

A constant pointer is declared as follows :

`Data_Type * const Pointer_Name;`

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

```
#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 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;
```

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;
}
```

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.
-

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.

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 .

Can static variables be declared in a header file?

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

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

Declaration of variable in c

A variable declaration only provides sureness to the compiler at the compile time that variable (<https://aticleworld.com/variable-in-c-language/>) 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 to the compiler that this variable may be defined within the function or outside of the function.

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

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

Definition of variable in c

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

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

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

What is the difference between pass by value by reference in c and pass by reference in c?

Pass By Value:

- & In this method value of the variable is passed. Changes made to formal will not affect the actual parameters.
- & Different memory locations will be created for both variables.
- & Here there will be temporary variable created in the function stack which does not affect the original variable.

Pass By Reference :

- & In Pass by reference, an address of the variable is passed to a function.
- & Whatever changes made to the formal parameter will affect the value of actual parameters(a variable whose address is passed).

- & Both formal and actual parameter shared the same memory location.
- & it is useful when you required to returns more than 1 values.

Aticleworld invites you to try skillshare (Unlimited Access to over 20,000 classes) Premium free for 2 months.



What is a reentrant function?

In computing, a computer program or subroutine is called reentrant if it can be interrupted in the middle of its execution and then safely be called again (“re-entered”) before its previous invocations complete execution. The interruption could be caused by an internal action such as a jump or call, or by an external action such as an interrupt or signal. Once the reentered invocation completes, the previous invocations will resume correct execution.

What is the inline function?

An inline keyword is a compiler directive that only suggests the compiler to substitute the body of the function at the calling the place. It is an optimization technique used by the compilers to reduce the overhead of function calls.

for example,

```
static inline void Swap(int *a, int *b)
{
    int tmp= *a;
    *a= *b;
    *b = tmp;
}
```

What is the advantage and disadvantage of the inline function?

There are few important advantage and disadvantage of the inline function.

Advantages:-

- 1)It saves the function calling overhead.
- 2) It also saves the overhead of variables push/pop on the stack, while function calling.
- 3) It also saves the overhead of return call from a function.
- 4) It increases locality of reference by utilizing instruction cache.
- 5) After inlining compiler can also apply intraprocedural optimization if specified. This is the most important one, in this way compiler can now focus on dead code elimination, can give more stress on branch prediction, induction variable elimination etc..

Disadvantages:-

- 1) May increase function size so that it may not fit in the cache, causing lots of cache miss.
 - 2) After inlining function, if variables number which are going to use register increases then they may create overhead on register variable resource utilization.
 - 3) It may cause compilation overhead as if somebody changes code inside an inline function then all calling location will also be compiled.
 - 4) If used in the header file, it will make your header file size large and may also make it unreadable.
 - 5) If somebody used too many inline functions resultant in a larger code size than it may cause thrashing in memory. More and number of page fault bringing down your program performance.
 - 6) It's not useful for an embedded system where large binary size is not preferred at all due to memory size constraints.
-

What is the virtual memory?

The virtual memory is the part of memory management techniques and it creates an illusion that the system has a sufficient amount of memory. In another word you can say that virtual memory is a layer of indirection.

Consider the two statements and find the difference between them?

```
struct sStudentInfo { char
Name[12];
int Age; float
Weight;
int RollNumber;

};

#define STUDENT_INFO struct sStudentInfo*
typedef struct sStudentInfo* studentInfo;

statement 1
STUDENT_INFO p1, p2;

statement 2 studentInfo
q1, q2;
```

Both statements look the same but actually, both are different to each other.

Statement 1 will be expanded to `struct sStudentInfo * p1, p2`. It means that `p1` is a pointer to `struct sStudentInfo` but `p2` is a variable of `struct sStudentInfo`.

In statement 2, both `q1` and `q2` will be a pointer to `struct sStudentInfo`.

What are the limitations of I2C interface?

- & Half duplex communication, so data is transmitted only in one direction (because of the single data bus) at a time.
- & Since the bus is shared by many devices, debugging an I2C bus (detecting which device is misbehaving) for issues is pretty difficult.
- & The I2C bus is shared by multiple slave devices if anyone of these slaves misbehaves (pull either SCL or SDA low for an indefinite time) the bus will be stalled. No further communication will take place.

- & I2C uses resistive pull-up for its bus. Limiting the bus speed.
 - & Bus speed is directly dependent on the bus capacitance, meaning longer I2C bus traces will limit the bus speed.
-

Here, I have mentioned some more embedded c interview questions for you. If you know, please write in the comment box. Might be your comment helpful for others.

- & What is the difference between flash memory, EPROM, and EEPROM?
- & What is the difference between Volatile & Non Volatile Memory?
- & What are the differences between a union and a structure in C?
- & What is the difference between RS232 and UART?
- & Is it possible to declare struct and union one inside other? Explain with example.
- & How to find the bug in code using the debugger if the pointer is pointing to an illegal value.
- & What is watchdog timer?
- & What is the DMA?
- & What is RTOS?
- & What are CAN and its uses?
- & Why is CAN having 120 ohms at each end?
- & Why is CAN message-oriented protocol?
- & What is the Arbitration in the CAN?
- & Standard CAN and Extended CAN difference?
- & What is the use of bit stuffing?
- & How many types of IPC mechanism do you know?
- & What is semaphore?
- & What is the spinlock?
- & Convert a given decimal number to hex.
- & What is the difference between heap and stack memory?
- & What is socket programming?
- & How can a double pointer be useful?
- & What is the difference between binary semaphore and mutex?

