# Volatile keyword/qulifire in c

# Volatile is a keyword in c. which tell the complier to avoided optimization on that variable

# When we are accessing any variable that time control go to address of that variable in memory and fetch the value in CPU and process on it, this process take some time to process.

# When we are accessing any variable again and again (like share variable and data coming from port) , and if the optimization is not disable then to speed-up the code execution complier make the copy of that variable in CPU register and process on that, instead of going to original place of the variable.

# So this is problem when we accessing data which is coming from port or when multiple thread accessing same global variable, and if complier make that variable as register then we may not get latest value of the variable.

# When we declare any variable as volatile then complier always read its value from its original place.

# So volatile keyword is used to tell complier that the value of the variable change any time so read its value from its original location instead of chase this variable value. Eg.volatile int a;

**Proper Use of C's volatile Keyword**

1. Memory-mapped peripheral registers

2. Global variables modified by an interrupt service routine

3. Global variables accessed by multiple tasks within a multi-threaded application

**Const** **Volatile Qualifier:-**

Const Volatile means that the program cannot modify the variable's value, but the value can be modified from the outside, thus no optimisations will be performed on the variable

**Not volatile means --value of variable cannot be changed by any external device or hardware**

**interrupt.**

What is meaning of the declaration:

const volatile int a=6;

Answer:

Value of variable cannot be changed by program (due to const) but its value can be changed by external device or hardware interrupt (due to volatile)

1. [**Difference between const & const volatile**](http://stackoverflow.com/questions/4592762/difference-between-const-const-volatile)

**Const:**

**When any variable has qualified with const keyword in declaration statement then it is not possible to assign any value or modify it by the program. But indirectly with the help of pointer its value can be changed.**Its meaning is that value of variable can be changed after the declaration statement by the program.   
#include<stdio.h>  
int main(){

const int a=5;

a++;

printf(“%d”,a);

return 0;

}

Output: Compiler error, we cannot modify const variable

# Static (variable) keyword/qualifier in c

# We can define any variable as static using static keyword.

# We use static keyword to limit the scope of variable within the file/block.

# Syntax:

# static datatype var\_name = var\_value;

1. A static int variable remains in memory while the program is running. A normal or auto variable is destroyed when a function call where the variable was declared is over.

For example, we can use static int to count number of times a function is called, but an auto variable can’t be sued for this purpose.

For example below program prints “1 2”

|  |
| --- |
| #include<stdio.h>  int fun()  {    static int count = 0;    count++;    return count;  }    int main()  {    printf("%d ", fun());    printf("%d ", fun());   return 0; } |

# Static variables are allocated memory in data segment, not stack segment.

# In C, static variables can only be initialized using constant literals. For example, following program fails in compilation

|  |
| --- |
| #include<stdio.h>  int initializer(void)  {      return 50;  }    int main()  {      static int i = initializer();      printf(" value of i = %d", i);      getchar();      return 0;  } |

Run on IDE

Output

In function 'main':

9:5: error: initializer element is not constant

static int i = initializer();

^

# Static global variables and functions are also possible in C/C++. The purpose of these is to limit scope of a variable or function to a file.

# !-What happen if we declare local variable as static ?

# -IF We declare local variable as static then

# i> scope is not change it is same within block

# but ii> life of that variable change stack to entire program

# iii> Static variables are allocated memory in data segment, not stack segment

# !! Static functions in C/WHAT HAPPEN IF WE

# DECLEAR FUNCTION AS STATIC ?

# In C, functions are global by default. The “*static*” keyword before a function name makes it static. For example, below function *fun()*is static.

|  |
| --- |
| static int fun(void){  printf("I am a static function ");} |

# Run on IDE

# Unlike global functions in C, access to static functions is restricted to the file where they are declared. Therefore, when we want to restrict access to functions, we make them static. Another reason for making functions static can be reuse of the same function name in other files.

# Const keyword in c (what is constant in c)

# If a variable declare as constant ,then we cannot use any operator on that variable Which can modify the value of the variable.

# E.g. =, ++ , -- , +=, - = etc.

# Using any of above assignment operator on constant variable (pointer also) cause complier time error.

# The qualifier const can be applied to the declaration of any variable to specify that its value will not be changed ( Which depends upon where const variables are stored, we may change value of const variable by using pointer ).

# Syntax:

# Const dataType variable name =value;

# E.g. const int a=10;

# i>we have to initialize constant variable at declaration time only.

# Ii>So the const variable use as read only variable

# Const int a=10;

# A is constant so any of the following statement will cause error message at compile time

# a++;

# a---;

# a=5;

# 

# But we can change constant variable value using pointer

# 

# Constant are used to declear constant pointer so that pointer value not modify

# 2) Pointer to constant.

# Pointer to constant can be declared in following two ways.

# const int \*ptr;

# or

# int const \*ptr;

# We can change pointer to point to any other integer variable, but cannot change value of object (entity) pointed using pointer ptr. Pointer is stored in read-write area (stack in present case). Object pointed may be in read only or read write area. Let us see following examples.

# #include <stdio.h>

# int main(void)

# {

# int i = 10;

# int j = 20;

# const int \*ptr = &i; /\* ptr is pointer to constant \*/

# printf("ptr: %d\n", \*ptr);

# \*ptr = 100; /\* error: object pointed cannot be modified

# using the pointer ptr \*/

# ptr = &j; /\* valid \*/

# printf("ptr: %d\n", \*ptr);

# return 0;

# }

# Output: error: assignment of read-only location ‘\*ptr’

**3) Constant pointer to variable.**

|  |
| --- |
| int \*const ptr; |

Run on IDE

Above declaration is constant pointer to integer variable, means we can change value of object pointed by pointer, but cannot change the pointer to point another variable.

|  |
| --- |
| #include <stdio.h>    int main(void)  {     int i = 10;     int j = 20;     int \*const ptr = &i;    /\* constant pointer to integer \*/       printf("ptr: %d\n", \*ptr);       \*ptr = 100;    /\* valid \*/     printf("ptr: %d\n", \*ptr);       ptr = &j;        /\* error \*/     return 0;  } |

Run on IDE

Output:

error: assignment of read-only variable ‘ptr’

**4) constant pointer to constant**

|  |
| --- |
| const int \*const ptr; |

Run on IDE

Above declaration is constant pointer to constant variable which means we cannot change value pointed by pointer as well as we cannot point the pointer to other variable. Let us see with example.

|  |
| --- |
| #include <stdio.h>    int main(void)  {      int i = 10;      int j = 20;      const int \*const ptr = &i;        /\* constant pointer to constant integer \*/        printf("ptr: %d\n", \*ptr);        ptr = &j;            /\* error \*/      \*ptr = 100;        /\* error \*/        return 0;  } |

Run on IDE

Output:

error: assignment of read-only variable ‘ptr’

error: assignment of read-only location ‘\*ptr’

# Typedef in c (what is typedef in c)

# Typedef is complier token. The preprocessor does not care about this

# Typedef is a keyword used to give a new name to existing data type.

#include<stdio.h>

**int** main()

{

**typedef** **int** Roll;

Roll num1 = 40,num2 = 20;

printf("Roll number 1 : %d",num1);

printf("Roll number 2 : %d",num2);

**return**(0);}

### typedef in C

typedef is a keyword used in C language to assign alternative names to existing datatypes. Its mostly used with user defined datatypes, when names of the datatypes become slightly complicated to use in programs. Following is the general syntax for using typedef,

typedef <existing\_name> <alias\_name>

Lets take an example and see how typedef actually works.

typedef unsigned long ulong;

The above statement define a term ulong for an unsigned long datatype. Now this ulong identifier can be used to define unsigned long type variables.

ulong i, j;

### Application of typedef

typedef can be used to give a name to user defined data type as well. Lets see its use with structures.

typedef struct

{

type member1;

type member2;

type member3;

} type\_name;

Here **type\_name** represents the stucture definition associated with it. Now this **type\_name** can be used to declare a variable of this stucture type.

type\_name t1, t2;

#### Example of Structure definition using typedef

#include<stdio.h>

#include<string.h>

typedef struct employee

{

char name[50];

int salary;

}emp;

void main( )

{

emp e1;

printf("\nEnter Employee record:\n");

printf("\nEmployee name:\t");

scanf("%s", e1.name);

printf("\nEnter Employee salary: \t");

scanf("%d", &e1.salary);

printf("\nstudent name is %s", e1.name);

printf("\nroll is %d", e1.salary);

}

### typedef and Pointers

typedef can be used to give an alias name to pointers also. Here we have a case in which use of typedef is beneficial during pointer declaration.

In Pointers \* binds to the right and not on the left.

int\* x, y;

By this declaration statement, we are actually declaring x as a pointer of type int, whereas y will be declared as a plain int variable.

typedef int\* IntPtr;

IntPtr x, y, z;

But if we use typedef like we have used in the example above, we can declare any number of pointers in a single statement.

# Difference between typedef and #define

***Typedef is different from #define among the following aspects***

* typedef is limited to giving symbolic names to types only, whereas #define can be used to define alias for values as well, e.g., you can define 1 as ONE, 3.14 as PI, etc.
* Typedef interpretation is performed by the compiler where #define statements are performed by preprocessor.
* #define should not be terminated with semicolon, but typedef should be terminated with semicolon.
* #define will just copy-paste the definition values at the point of use, while typedef is actual definition of a new type.
* Typedef follows the scope rule which means if a new type is defined in a scope (inside a function), then the new type name will only be visible till the scope is there. In case of #define, when preprocessor encounters #define, it replaces all the occurrences, after that (No scope rule is followed).

# Difference between “typedef”’ and “MACRO”

1. macro is a preprocessor directive start with # define so it is preproccer token
2. typedef is complier token
3. Typedef defines a new data type.

Eg. typedef int number;

Number a=10;

1. Macros can be of any type. Macros can even be any code block containing statements, loops, function calls etc.

a #define WORD unsigned short

#define MACRO(num, str) ({\

            printf("%d", num);\

            printf(" is");\

            printf(" %s number", str);\

            printf("\n");\     })

1. #define should not be terminated with semicolon, but typedef should be terminated with semicolon.
2. inline function:

* What is inline function?

This feature available in only “C” and “C++” other programming language don’t have this funcanality.

***Definition: inline int max (int x, int y)***

***Inline function is a simple function that is used to save time which is wasted in navigating from current function to called function and return to calling function.***

1. ***What will happen when you call a function in “c” Programming? The function***

***Will not be call in one shot, there are lot of internal sequences that will go***

***Ahead and happened first.***

1. ***First step when we going to call we should know that Where I return after execution of the function***
2. ***So I need to store return address, first in the stack of calling function.***
3. ***When called function executed and control come back in main function for ferther execution we need to store address where I going to jump(Address of the next***

***Instruction of the calling function (that is an instruction after function call)***

***In program counter when the execution come back to stack take return address and***

***Navigate back to main function***

***This is like I need to go next house🡪but before going next house I need to know that I should come hack in house where I start –so I take address where I will return.-->to do this it need to do lot of background stuff (like create stack ,assign memory for function local variable, changing programming counter and registers )***

***The point is simple but it need lot of background process to complete this***

***So how long will it take for one function to call in c code, it will take minimal time but how will it take when we call same function so many time say 100 OR 1000 time?***

***It west lot of time for moving and coming so instead of that***

***If you just keep the minimal code into main program itself still maintaining the modularity it will be better.***

***The inline function tell complier whenever there is necessity just copy paste (inline function) code into main code itself.***

***So you will not wastage time ingoing and coming it will be the time you saving.***

***So main ambition of inline function is to save the time that you are going to waste in navigating from main function to called function & back from called function to main function.***

***--->Inline is not an order it is a request to compiler.***

***When you declare a function inline the complier will looking into function to see is it really feasible to make that function inline?***

***If it is not feasible it wouldn’t make it inline.***

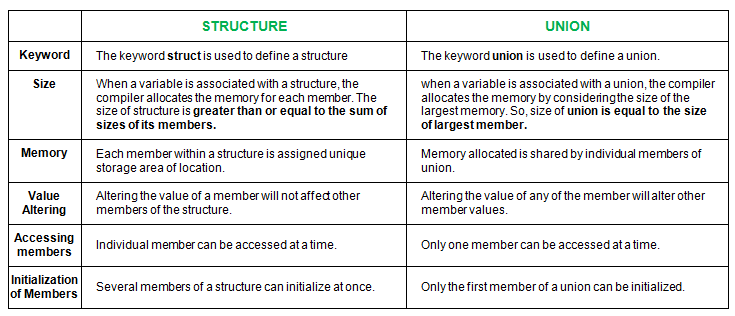
1. Difference between macro and inline function?
2. ***Difference***

|  |  |  |
| --- | --- | --- |
|  | ***Inline function*** | ***Macro*** |
| ***Keyword*** | ***The keyword inline is used to define a function*** | ***The keyword #define is used to define a macro*** |
| ***Memory*** | ***Function code is copied in calling function code, so the size of the pregame is increase*** | ***Micro name is replaced with its original code. Memory size increase*** |
| ***Syntax checking*** | ***Type checking is done here***  ***Inline int max (int x, int y)*** | ***It doesn’t support Type checking***  ***#define max(x,y)*** |
| ***Token*** | ***Complier time*** | ***Preprocessor directive*** |
| ***Completion*** | ***It is not an order it is only request to compiler, complier will decide to make it inline or not.***  ***If feasible then complier copy the function code into calling function*** | ***It is preprocessor token. Preprocessor replace macro body whereas macro name is written.*** |

# Difference Between malloc and calloc

| **BASIS OF COMPARISON** | **MALLOC()** | **CALLOC()** |
| --- | --- | --- |
| No of blocks | Allocates an only single block of requested memory. | Allocates multiple blocks of the requested memory. |
| Syntax | void \*malloc(size\_t size); | void \*calloc(size\_t num, size\_t size); |
| Initialization | malloc() doesn't clear and initialize the allocated memory. | calloc() initializes the allocated memory to zero. |
| Manner of Allocation | malloc() function allocates memory of size 'size' from the heap. | calloc() function allocates memory the size of which is equal to num \*size. |
| Speed | Fast | Comparatively slow. |

1. Difference between Structure and Union in C



|  |  |
| --- | --- |
| A structure is a user-defined data type available in C that allows to combining data items of different kinds. Structures are used to represent a record | A union is a special data type available in C that allows storing different data types in the same memory location. You can define a union with many members, but only one member can contain a value at any given time. Unions provide an efficient way of using the same memory location for multiple purposes. |

**Similarities between Structure and Union**

1. ***Both are user-defined data types used to store data of different types as a single unit.***
2. ***Their members can be objects of any type, including other structures and unions or arrays. A member can also consist of a bit field.***
3. ***Both structures and unions support only assignment = and sizeof operators. The two structures or unions in the assignment must have the same members and member types.***
4. ***A structure or a union can be passed by value to functions and returned by value by functions. The argument must have the same type as the function parameter. A structure or union is passed by value just like a scalar variable as a corresponding parameter.***

# 5> Interesting Facts about Macros and Preprocessors in C

***In a C program, all lines that start with # are processed by preprocessor which is a special program invoked by the compiler. In a very basic term, preprocessor takes a C program and produces another C program without any #.***

***Following are some interesting facts about preprocessors in C.***

***1) When we use include directive, the contents of included header file (after preprocessing) are copied to the current file.  
Angular brackets < and > instruct the preprocessor to look in the standard folder where all header files are held.  Double quotes “and “instruct the preprocessor to look into the current folder and if the file is not present in current folder, then in standard folder of all header files.***

***2) When we use define for a constant, the preprocessor produces a C program where the defined constant is searched and matching tokens are replaced with the given expression. For example in the following program max is defined as 100.***

|  |
| --- |
| ***#include<stdio.h>***  ***#define max 100***  ***int main()***  ***{***  ***printf("max is %d", max);***  ***return 0;***  ***}***  ***// Output: max is 100***  ***// Note that the max inside "" is not replaced*** |

# Difference between function call by value and call by reference in c

|  |  |  |
| --- | --- | --- |
| **No.** | **Call by value** | **Call by reference** |
| 1 | A copy of value is passed to the function | An address of value is passed to the function |
| 2 | Changes made inside the function is not reflected on other functions | Changes made inside the function is reflected outside the function also |
| 3 | Actual and formal arguments will be created in different memory location | Actual and formal arguments will be created in same memory location |

## Difference between macro and function

|  |  |  |
| --- | --- | --- |
| **No** | **Macro** | **Function** |
| 1 | Macro is **Preprocessed** | Function is **Compiled** |
| 2 | **No Type Checking** | **Type Checking** is Done |
| 3 | **Code** Length **Increases** | **Code** Length remains **Same** |
| 4 | Use of macro can lead to **side effect** | No **side Effect** |
| – |  | |
| 5 | Speed of Execution is **Faster** | Speed of Execution is **Slower** |
| 6 | Before Compilation macro name is replaced by macro value | During function call , Transfer of Control takes place |
| 7 | Useful where small code appears many time | Useful where large code appears many time |
| 8 | Generally Macros do not extend beyond one line | Function can be of any number of lines |
| 9 | Macro does not Check **Compile Errors** | Function Checks **Compile Errors** |

Here are the differences between macro and function,

**1. Macro consumes less time:**  
When a function is called, arguments have to be passed to it, those arguments are accepted by corresponding dummy variables in the function. Then they are processed, and finally the function returns a value that is assigned to a variable (except for a void function). If a function is invoked number of times, the times add up, and compilation is delayed. On the other hand, the macro expansion had already taken place and replaced each occurrence of the macro in the source code before the source code starts compiling, so it requires no additional time to execute.

**2. Function consumes less memory**:  
Prior to compilation, all the macro-presences are replaced by their corresponding macro expansions, which consumes considerable memory. On the other hand, even if a function is invoked 100 times, it still occupies the same space. Hence function consumes less memory.

# 9. difference between Volatile and Constant

|  |
| --- |
| The difference between Volatile and Const can be easily see in bellow case,  1) If you say some variable as Const, it may not be possible to modify by your program.  2) if you say volatile, it is just giving a hint to the compiler not to optimize the code, because  the value may be changed from the external threads or other programs.  3) if we define a variable as Const Volatile, that means this variable can not be modified by  same program, will not be optimized by compiler and can be modified by external threads  or programs.  example:  if i write a function like below,  const freq = 10;  calfreq()  {  return (Const freq \* 2);  }  here in this case compiler may optimize the code to  return(20);  all the time.  But here in m y case, freq value may change, because of external hardware / threads / programs So, if i say Const Volatile, then problem will be fixed.  =================================================================== |

# C – Preprocessor directives

#### [PREV](http://fresh2refresh.com/c/c-union/)     [NEXT](http://fresh2refresh.com/c-programming/c-file-handling/)

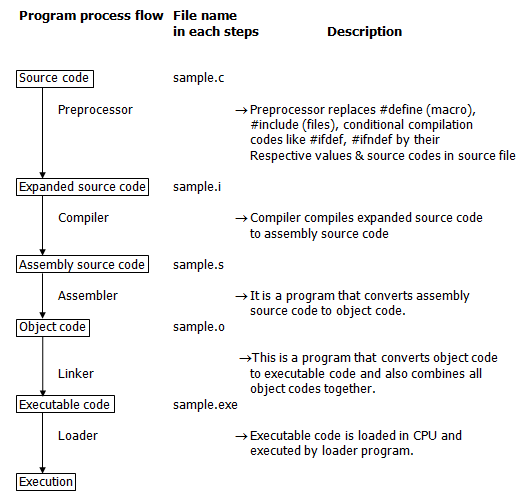
#### **C PREPROCESSOR DIRECTIVES:**

* Before a C program is compiled in a compiler, source code is processed by a program called preprocessor. This process is called preprocessing.
* Commands used in preprocessor are called preprocessor directives and they begin with “#” symbol.

Below is the list of preprocessor directives that C programming language offers.

|  |  |
| --- | --- |
| **Preprocessor** | **Syntax/Description** |
| Macro | **Syntax:**#defineThis macro defines constant value and can be any of the basic data types. |
| Header file inclusion | **Syntax:**#include <file\_name> The source code of the file “file\_name” is included in the main program at the specified place. |
| Conditional compilation | **Syntax:**#ifdef, #endif, #if, #else, #ifndefSet of commands are included or excluded in source program before compilation with respect to the condition. |
| Other directives | **Syntax:**#undef, #pragma #undef is used to undefine a defined macro variable. #Pragma is used to call a function before and after main function in a C program. |

A program in C language involves into different processes. Below diagram will help you to understand all the processes that a C program comes across.



There are 4 regions of memory which are created by a compiled C program. They are,

1. **First region** – This is the memory region which holds the executable code of the program.
2. **2nd region**  – In this memory region, global variables are stored.
3. **3rd region**   – stack
4. **4th region**   – heap

#### **DO YOU KNOW DIFFERENCE BETWEEN STACK & HEAP MEMORY IN C LANGUAGE?**

|  |  |
| --- | --- |
| **Stack** | **Heap** |
| Stack is a memory region where “local variables”, “return addresses of function calls” and “arguments to functions” are hold while C program is executed. | Heap is a memory region which is used by dynamic memory allocation functions at run time. |
| CPU’s current state is saved in stack memory | Linked list is an example which uses heap memory. |

# 6. Difference Between Semaphore and Mutex

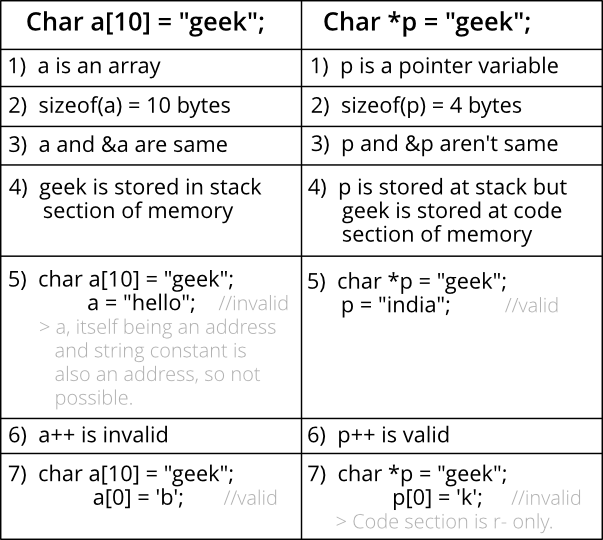
| **BASIS FOR COMPARISON** | **SEMAPHORE** | **MUTEX** |
| --- | --- | --- |
| Basic | Semaphore is a signalling mechanism. | Mutex is a locking mechanism. |
| Existence | Semaphore is an integer variable. | Mutex is an object. |
| Function | Semaphore allow multiple program threads to access a finite instance of resources. | Mutex allow multiple program thread to access a single resource but not simultaneously. |
| Ownership | Semaphore value can be changed by any process acquiring or releasing the resource. | Mutex object lock is released only by the process that has acquired the lock on it. |
| Categorize | Semaphore can be categorized into counting semaphore and binary semaphore. | Mutex is not categorized further. |
| Operation | Semaphore value is modified using wait() and signal() operation. | Mutex object is locked or unlocked by the process requesting or releasing the resource. |
| Resources |  |  |

# What’s difference between char s[] and char \*s in C?

Consider below two statements in C. What is difference between two?

char s[] = "geeksquiz";

char \*s = "geeksquiz";

**Below are the key differences:**  
  


The statements ‘**char s[] = “geeksquiz”**‘ creates a character array which is like any other array and we can do all array operations. The only special thing about this array is, although we have initialized it with 9 elements, its size is 10 (Compiler automatically adds ‘\0’)

filter\_none

edit

play\_arrow

brightness\_4

|  |
| --- |
| #include <stdio.h>  int main()  {      char s[] = "geeksquiz";      printf("%lu", sizeof(s));      s[0] = 'j';      printf("\n%s", s);      return 0;  } |

Output:

10

jeeksquiz

The statement ‘**char \*s = “geeksquiz”**‘ creates a string literal. The string literal is stored in read only part of memory by most of the compilers. The C and C++ standards say that string literals have static storage duration, any attempt at modifying them gives undefined behavior.  
**s** is just a pointer and like any other pointer stores address of string literal.

filter\_none

edit

play\_arrow

brightness\_4

|  |
| --- |
| #include <stdio.h>  int main()  {      char \*s = "geeksquiz";      printf("%lu", sizeof(s));        // Uncommenting below line would cause undefined behaviour      // (Caused segmentation fault on gcc)      //  s[0] = 'j';      return 0;  } |

Output:

8

Running above program may generates a warning also “warning: deprecated conversion from string constant to ‘char\*’”. This warning occurs because s is not a const pointer, but stores address of read only location. The warning can be avoided by pointer to const.

filter\_none

edit

play\_arrow

brightness\_4

|  |
| --- |
| #include <stdio.h>  int main()  {      const char \*s = "geeksquiz";      printf("%lu", sizeof(s));      return 0;  } |

This article is contributed by Abhay Rathi. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above