Contents

[Bit Fields in C 2](#_Toc528328446)

[Command line arguments in C/C++ 6](#_Toc528328447)

[Compiling a C program:- Behind the Scenes 8](#_Toc528328448)

[How do we compile and run a C program? 8](#_Toc528328449)

[What goes inside the compilation process? 9](#_Toc528328450)

[Pre-processing 9](#_Toc528328451)

[Compiling 9](#_Toc528328452)

[Assembly 10](#_Toc528328453)

[Linking 10](#_Toc528328454)

[Const Qualifier in C 10](#_Toc528328455)

[2) Pointer to constant. 11](#_Toc528328456)

[**3) Constant pointer to variable.** 13](#_Toc528328457)

[**4) constant pointer to constant** 13](#_Toc528328458)

[Dangling, Void , Null and Wild Pointers 14](#_Toc528328459)

[**Dangling pointer** 14](#_Toc528328460)

[**Void pointer** 16](#_Toc528328461)

[**NULL Pointer** 16](#_Toc528328462)

[**Wild pointer** 17](#_Toc528328463)

[Difference between Structure and Union in C 17](#_Toc528328464)

[**structures in C** 17](#_Toc528328465)

[**union** 17](#_Toc528328466)

[**Similarities between Structure and Union** 18](#_Toc528328467)

[**Differences** 18](#_Toc528328468)

[Function Pointer in C 21](#_Toc528328469)

[**Following are some interesting facts about function pointers.** 21](#_Toc528328470)

[Interesting Facts about Macros and Preprocessors in C 25](#_Toc528328471)

[Little and Big Endian Mystery 29](#_Toc528328472)

[What are these? 29](#_Toc528328473)

[Macros vs Functions 31](#_Toc528328474)

[Memory Layout of C Programs 33](#_Toc528328475)

[Storage Classes in C 36](#_Toc528328476)

[**auto** 36](#_Toc528328477)

[**extern** 36](#_Toc528328478)

[**static** 37](#_Toc528328479)

[**register** 37](#_Toc528328480)

[Structure Member Alignment, Padding and Data Packing 40](#_Toc528328481)

[Understanding “volatile” qualifier in C | Set 2 (Examples) 45](#_Toc528328482)

[calloc() versus malloc() 48](#_Toc528328483)

Bit Fields in C

In C, we can specify size (in bits) of structure and union members. The idea is to use memory efficiently when we know that the value of a field or group of fields will never exceed a limit or is withing a small range.

For example, consider the following declaration of date without use of bit fields.

|  |
| --- |
| #include <stdio.h>    // A simple representation of date  struct date  {     unsigned int d;     unsigned int m;     unsigned int y;  };    int main()  {     printf("Size of date is %d bytes\n", sizeof(struct date));     struct date dt = {31, 12, 2014};     printf("Date is %d/%d/%d", dt.d, dt.m, dt.y);  } |

Run on IDE

Output:

Size of date is 12 bytes

Date is 31/12/2014

The above representation of ‘date’ takes 12 bytes on a compiler where an unsigned int takes 4 bytes. Since we know that the value of d is always from 1 to 31, value of m is from 1 to 12, we can optimize the space using bit fields.

|  |
| --- |
| #include <stdio.h>    // A space optimized representation of date  struct date  {     // d has value between 1 and 31, so 5 bits     // are sufficient     unsigned int d: 5;       // m has value between 1 and 12, so 4 bits     // are sufficient     unsigned int m: 4;       unsigned int y;  };    int main()  {     printf("Size of date is %d bytes\n", sizeof(struct date));     struct date dt = {31, 12, 2014};     printf("Date is %d/%d/%d", dt.d, dt.m, dt.y);     return 0;  } |

Run on IDE

Output:

Size of date is 8 bytes

Date is 31/12/2014

**Following are some interesting facts about bit fields in C.**

**1)** A special unnamed bit field of size 0 is used to force alignment on next boundary. For example consider the following program.

|  |
| --- |
| #include <stdio.h>    // A structure without forced alignment  struct test1  {     unsigned int x: 5;     unsigned int y: 8;  };    // A structure with forced alignment  struct test2  {     unsigned int x: 5;     unsigned int: 0;     unsigned int y: 8;  };    int main()  {     printf("Size of test1 is %d bytes\n", sizeof(struct test1));     printf("Size of test2 is %d bytes\n", sizeof(struct test2));     return 0;  } |

Run on IDE

Output:

Size of test1 is 4 bytes

Size of test2 is 8 bytes

**2)** We cannot have pointers to bit field members as they may not start at a byte boundary.

|  |
| --- |
| #include <stdio.h>  struct test  {     unsigned int x: 5;     unsigned int y: 5;     unsigned int z;  };  int main()  {     struct test t;       // Uncommenting the following line will make     // the program compile and run     printf("Address of t.x is %p", &t.x);       // The below line works fine as z is not a     // bit field member     printf("Address of t.z is %p", &t.z);     return 0;  } |

Run on IDE

Output:

error: attempt to take address of bit-field structure member 'test::x'

**3)**It is implementation defined to assign an out-of-range value to a bit field member.

|  |
| --- |
| #include <stdio.h>  struct test  {     unsigned int x: 2;     unsigned int y: 2;     unsigned int z: 2;  };  int main()  {     struct test t;     t.x = 5;     printf("%d", t.x);     return 0;  } |

Run on IDE

Output:

Implementation-Dependent

**4)** In C++, we can have static members in a structure/class, but bit fields cannot be static.

|  |
| --- |
| // The below C++ program compiles and runs fine  struct test1 {     static unsigned int x;  };  int main() {  }      // But below C++ program fails in compilation as bit fields  // cannot be static  struct test1 {     static unsigned int x: 5;  };  int main() {  }  // error: static member 'x' cannot be a bit-field |

Run on IDE

**5)** Array of bit fields is not allowed. For example, the below program fails in compilation.

|  |
| --- |
| struct test  {    unsigned int x[10]: 5;  };    int main()  {    } |

Run on IDE

Output:

error: bit-field 'x' has invalid type

**Exercise:**  
Predict the output of following programs. Assume that unsigned int takes 4 bytes and long int takes 8 bytes.  
**1)**

|  |
| --- |
| #include <stdio.h>  struct test  {     unsigned int x;     unsigned int y: 33;     unsigned int z;  };  int main()  {     printf("%d", sizeof(struct test));     return 0;  } |

Run on IDE

**2)**

|  |
| --- |
| #include <stdio.h>  struct test  {     unsigned int x;     long int y: 33;     unsigned int z;  };  int main()  {     struct test t;     unsigned int \*ptr1 = &t.x;     unsigned int \*ptr2 = &t.z;     printf("%d", ptr2 - ptr1);     return 0;  } |

Run on IDE

3)

|  |
| --- |
| union test  {    unsigned int x: 3;    unsigned int y: 3;    int z;  };    int main()  {     union test t;     t.x = 5;     t.y = 4;     t.z = 1;     printf("t.x = %d, t.y = %d, t.z = %d",             t.x, t.y, t.z);     return 0;  } |

Run on IDE

**4)** Use bit fields in C to figure out a way whether a machine is little endian or big endian.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

# Command line arguments in C/C++

The most important function of C/C++ is main() function. It is mostly defined with a return type of int and without parameters :

int main() { /\* ... \*/ }

We can also give command-line arguments in C and C++. Command-line arguments are given after the name of the program in command-line shell of Operating Systems.  
To pass command line arguments, we typically define main() with two arguments : first argument is the number of command line arguments and second is list of command-line arguments.

int main(int argc, char \*argv[]) { /\* ... \*/ }

or

int main(int argc, char \*\*argv) { /\* ... \*/ }

* **argc (ARGument Count)** is int and stores number of command-line arguments passed by the user including the name of the program. So if we pass a value to a program, value of argc would be 2 (one for argument and one for program name)
* The value of argc should be non negative.
* **argv(ARGument Vector)** is array of character pointers listing all the arguments.
* If argc is greater than zero,the array elements from argv[0] to argv[argc-1] will contain pointers to strings.
* Argv[0] is the name of the program , After that till argv[argc-1] every element is command -line arguments.

For better understanding run this code on your linux machine.

|  |
| --- |
| // Name of program mainreturn.cpp  #include <iostream>  using namespace std;    int main(int argc, char\*\* argv)  {      cout << "You have entered " << argc           << " arguments:" << "\n";        for (int i = 0; i < argc; ++i)          cout << argv[i] << "\n";        return 0;  } |

Run on IDE

Input:

$ g++ mainreturn.cpp -o main

$ ./main geeks for geeks

Output:

You have entered 4 arguments:

./main

geeks

for

geeks

**Note :** Other platform-dependent formats are also allowed by the C and C++ standards; for example, Unix (though not POSIX.1) and Microsoft Visual C++ have a third argument giving the program’s environment, otherwise accessible through getenv in stdlib.h: Refer [C program to print environment variables](http://quiz.geeksforgeeks.org/c-program-print-environment-variables/) for details.

**Properties of Command Line Arguments:**

1. They are passed to main() function.
2. They are parameters/arguments supplied to the program when it is invoked.
3. They are used to control program from outside instead of hard coding those values inside the code.
4. argv[argc] is a NULL pointer.
5. argv[0] holds the name of the program.
6. argv[1] points to the first command line argument and argv[n] points last argument.

**Note :** You pass all the command line arguments separated by a space, but if argument itself has a space then you can pass such arguments by putting them inside double quotes “” or single quotes ”.

|  |
| --- |
| // C program to illustrate  // command line arguments  #include<stdio.h>    int main(int argc,char\* argv[])  {      int counter;      printf("Program Name Is: %s",argv[0]);      if(argc==1)          printf("\nNo Extra Command Line Argument Passed Other Than Program Name");      if(argc>=2)      {          printf("\nNumber Of Arguments Passed: %d",argc);          printf("\n----Following Are The Command Line Arguments Passed----");          for(counter=0;counter<argc;counter++)              printf("\nargv[%d]: %s",counter,argv[counter]);      }      return 0;  } |

Run on IDE

**Output in different scenarios:**

1. **Without argument:** When the above code is compiled and executed without passing any argument, it produces following output.
2. $ ./a.out
3. Program Name Is: ./a.out
4. No Extra Command Line Argument Passed Other Than Program Name
5. **Three arguments :** When the above code is compiled and executed with a three arguments, it produces the following output.
6. $ ./a.out First Second Third
7. Program Name Is: ./a.out
8. Number Of Arguments Passed: 4
9. ----Following Are The Command Line Arguments Passed----
10. argv[0]: ./a.out
11. argv[1]: First
12. argv[2]: Second
13. argv[3]: Third
14. **Single Argument :** When the above code is compiled and executed with a single argument separated by space but inside double quotes, it produces the following output.
15. $ ./a.out "First Second Third"
16. Program Name Is: ./a.out
17. Number Of Arguments Passed: 2
18. ----Following Are The Command Line Arguments Passed----
19. argv[0]: ./a.out
20. argv[1]: First Second Third
21. **Single argument in quotes separated by space :**When the above code is compiled and executed with a single argument separated by space but inside single quotes, it produces the following output.
22. $ ./a.out 'First Second Third'
23. Program Name Is: ./a.out
24. Number Of Arguments Passed: 2
25. ----Following Are The Command Line Arguments Passed----
26. argv[0]: ./a.out

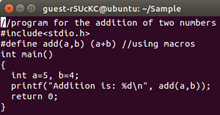
argv[1]: First Second Third

# Compiling a C program:- Behind the Scenes

C is a high level language and it needs a compiler to convert it into an executable code so that the program can be run on our machine.

## **How do we compile and run a C program?**

Below are the steps we use on an Ubuntu machine with gcc compiler.

* [](https://www.geeksforgeeks.org/wp-content/uploads/compilation.png)We first create a C program using an editor and save the file as filename.c

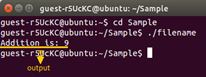
**$ vi filename.c**

The diagram on right shows a simple program to add two numbers.

* [](https://www.geeksforgeeks.org/wp-content/uploads/compil31.png)Then compile it using below command.

**$ gcc –Wall filename.c –o filename**

The option -Wall enables all compiler’s warning messages. This option is recommended to generate better code.  
The option -o is used to specify output file name. If we do not use this option, then an output file with name a.out is generated.

* [](https://www.geeksforgeeks.org/wp-content/uploads/compil21.png)After compilation executable is generated and we run the generated executable using below command.

**$ ./filename**

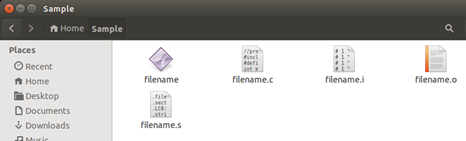
## **What goes inside the compilation process?**

Compiler converts a C program into an executable. There are four phases for a C program to become an executable:

1. Pre-processing
2. Compilation
3. Assembly
4. Linking

By executing below command, We get the all intermediate files in the current directory along with the executable.

**$gcc –Wall –save-temps filename.c –o filename**

The following screenshot shows all generated intermediate files.  
[](https://www.geeksforgeeks.org/wp-content/uploads/compil4.png)

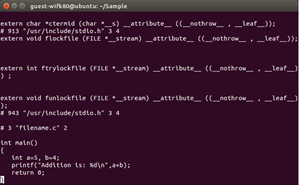
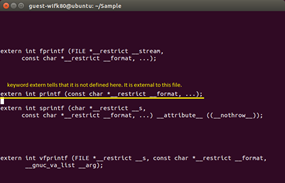
Let us one by one see what these intermediate files contain.

## **Pre-processing**

This is the first phase through which source code is passed. This phase include:

* Removal of Comments
* Expansion of Macros
* Expansion of the included files.

The preprocessed output is stored in the **filename.i**. Let’s see what’s inside filename.i: using **$vi filename.i**

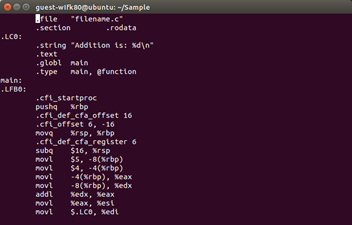
[](https://www.geeksforgeeks.org/wp-content/uploads/compil5.png)[](https://www.geeksforgeeks.org/wp-content/uploads/compile6.png)

In the above output, source file is filled with lots and lots of info, but at the end our code is preserved.  
**Analysis:**

* printf contains now a + b rather than add(a, b) that’s because macros have expanded.
* Comments are stripped off.
* **#include<stdio.h>** is missing instead we see lots of code. So header files has been expanded and included in our source file.

## **Compiling**

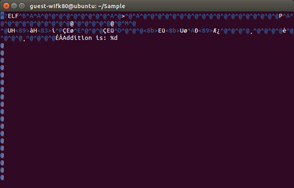
The next step is to compile filename.i and produce an; intermediate compiled output file **filename.s**. This file is in assembly level instructions. Let’s see through this file using**$vi filename.s**

[](https://www.geeksforgeeks.org/wp-content/uploads/image2.png)

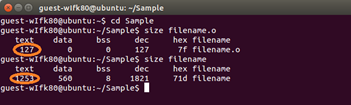
The snapshot shows that it is in assembly language, which assembler can understand.

## **Assembly**

In this phase the filename.s is taken as input and turned into **filename.o** by assembler. This file contain machine level instructions. At this phase, only existing code is converted into machine language, the function calls like printf() are not resolved. Let’s view this file using **$vi filename.o**

[](https://www.geeksforgeeks.org/wp-content/uploads/compil7.png)

## **Linking**

This is the final phase in which all the linking of function calls with their definitions are done. Linker knows where all these functions are implemented. Linker does some extra work also, it adds some extra code to our program which is required when the program starts and ends. For example, there is a code which is required for setting up the environment like passing command line arguments. This task can be easily verified by using **$size filename.o** and **$size filename**. Through these commands, we know that how output file increases from an object file to an executable file. This is because of the extra code that linker adds with our program.  
[](https://www.geeksforgeeks.org/wp-content/uploads/compil8.png)  
Note that GCC by default does dynamic linking, so printf() is dynamically linked in above program. Refer [this](https://www.geeksforgeeks.org/static-vs-dynamic-libraries/), [this](https://www.geeksforgeeks.org/working-with-shared-libraries-set-1/)and [this](https://www.geeksforgeeks.org/working-with-shared-libraries-set-2/) for more details on static and dynamic linkings.

This article is contributed by [**Vikash Kumar**](https://www.linkedin.com/in/vikash-kumar-2b947389). Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

# Const Qualifier in C

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 ). The result is implementation-defined if an attempt is made to change a const.  
1) Pointer to variable.

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

Run on IDE

We can change the value of ptr and we can also change the value of object ptr pointing to. Pointer and value pointed by pointer both are stored in read-write area. See the following code fragment.

|  |
| --- |
| #include <stdio.h>  int main(void)  {      int i = 10;      int j = 20;      int \*ptr = &i;        /\* pointer to integer \*/      printf("\*ptr: %d\n", \*ptr);        /\* pointer is pointing to another variable \*/      ptr = &j;      printf("\*ptr: %d\n", \*ptr);        /\* we can change value stored by pointer \*/      \*ptr = 100;      printf("\*ptr: %d\n", \*ptr);        return 0;  } |

Output:

\*ptr: 10

\*ptr: 20

\*ptr: 100

2) Pointer to constant.  
Pointer to constant can be declared in following two ways.

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

Run on IDE

or

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

Run on IDE

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;  } |

Run on IDE

Output:

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

Following is another example where variable i itself is constant.

|  |
| --- |
| #include <stdio.h>    int main(void)  {      int const i = 10;    /\* i is stored in read only area\*/      int j = 20;        int const \*ptr = &i;        /\* pointer to integer constant. Here i                                   is of type "const int", and &i is of                                   type "const int \*".  And p is of type                                  "const int", types are matching no issue \*/        printf("ptr: %d\n", \*ptr);        \*ptr = 100;        /\* error \*/        ptr = &j;          /\* valid. We call it as up qualification. In                           C/C++, the type of "int \*" is allowed to up                           qualify to the type "const int \*". The type of                           &j is "int \*" and is implicitly up qualified by                           the compiler to "cons tint \*" \*/        printf("ptr: %d\n", \*ptr);        return 0;  } |

Run on IDE

Output:

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

Down qualification is not allowed in C++ and may cause warnings in C. Following is another example with down qualification.

|  |
| --- |
| #include <stdio.h>    int main(void)  {      int i = 10;      int const j = 20;        /\* ptr is pointing an integer object \*/      int \*ptr = &i;        printf("\*ptr: %d\n", \*ptr);        /\* The below assignment is invalid in C++, results in error         In C, the compiler \*may\* throw a warning, but casting is         implicitly allowed \*/      ptr = &j;        /\* In C++, it is called 'down qualification'. The type of expression         &j is "const int \*" and the type of ptr is "int \*". The         assignment "ptr = &j" causes to implicitly remove const-ness         from the expression &j. C++ being more type restrictive, will not         allow implicit down qualification. However, C++ allows implicit         up qualification. The reason being, const qualified identifiers         are bound to be placed in read-only memory (but not always). If         C++ allows above kind of assignment (ptr = &j), we can use 'ptr'         to modify value of j which is in read-only memory. The         consequences are implementation dependent, the program may fail         at runtime. So strict type checking helps clean code. \*/        printf("\*ptr: %d\n", \*ptr);        return 0;  }    // Reference <http://www.dansaks.com/articles/1999-02%20const%20T%20vs%20T%20const.pdf>    // More interesting stuff on C/C++ @ <http://www.dansaks.com/articles.htm> |

## **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’

# Dangling, Void , Null and Wild Pointers

## **Dangling pointer**

A pointer pointing to a memory location that has been deleted (or freed) is called dangling pointer. There are **three** different ways where Pointer acts as dangling pointer

1. **De-allocation of memory**

|  |
| --- |
| // Deallocating a memory pointed by ptr causes  // dangling pointer  #include <stdlib.h>  #include <stdio.h>  int main()  {      int \*ptr = (int \*)malloc(sizeof(int));        // After below free call, ptr becomes a      // dangling pointer      free(ptr);        // No more a dangling pointer      ptr = NULL;  } |

1. Copy CodeRun on IDE
2. **Function Call**

|  |
| --- |
| // The pointer pointing to local variable becomes  // dangling when local variable is not static.  #include<stdio.h>    int \*fun()  {      // x is local variable and goes out of      // scope after an execution of fun() is      // over.      int x = 5;        return &x;  }    // Driver Code  int main()  {      int \*p = fun();      fflush(stdin);        // p points to something which is not      // valid anymore      printf("%d", \*p);      return 0;  } |

1. Copy CodeRun on IDE
2. Output:
3. A garbage Address
4. The above problem doesn’t appear (or p doesn’t become dangling) if x is a static variable.

|  |
| --- |
| // The pointer pointing to local variable doesn't  // become dangling when local variable is static.  #include<stdio.h>    int \*fun()  {      // x now has scope throughout the program      static int x = 5;      return &x;  }    int main()  {      int \*p = fun();      fflush(stdin);      // Not a dangling pointer as it points      // to static variable.      printf("%d",\*p);  } |

1. Copy CodeRun on IDE
2. Output:
3. 5
4. **Variable goes out of scope**
5. void main()
6. {
7. int \*ptr;
8. .....
9. .....
10. {
11. int ch;
12. ptr = &ch;
13. }
14. .....
15. // Here ptr is dangling pointer
16. }

## [**Void pointer**](http://quiz.geeksforgeeks.org/void-pointer-c/)

Void pointer is a specific pointer type – void \* – a pointer that points to some data location in storage, which doesn’t have any specific type. Void refers to the type. Basically the type of data that it points to is can be any. If we assign address of char data type to void pointer it will become char Pointer, if int data type then int pointer and so on. Any pointer type is convertible to a void pointer hence it can point to any value.  
**Important Points**

1. void pointers **cannot be dereferenced**. It can however be done using typecasting the void pointer
2. Pointer arithmetic is not possible on pointers of void due to lack of concrete value and thus size.

**Example:**

|  |
| --- |
| #include<stdlib.h>    int main()  {      int x = 4;      float y = 5.5;      //A void pointer      void \*ptr;      ptr = &x;      // (int\*)ptr - does type casting of void      // \*((int\*)ptr) dereferences the typecasted      // void pointer variable.      printf("Integer variable is = %d", \*( (int\*) ptr) );      // void pointer is now float      ptr = &y;      printf("\nFloat variable is= %f", \*( (float\*) ptr) );      return 0;  } |

Copy CodeRun on IDE

Output:

Integer variable is = 4

Float variable is= 5.500000

Refer [void pointer article](http://quiz.geeksforgeeks.org/void-pointer-c/) for details.

## [**NULL Pointer**](http://quiz.geeksforgeeks.org/few-bytes-on-null-pointer-in-c/)

NULL Pointer is a pointer which is pointing to nothing. In case, if we don’t have address to be assigned to a pointer, then we can simply use NULL.

|  |
| --- |
| #include <stdio.h>  int main()  {      // Null Pointer      int \*ptr = NULL;      printf("The value of ptr is %u", ptr);      return 0;  } |

Copy CodeRun on IDE

Output :

The value of ptr is 0

**Important Points**

1. **NULL vs Uninitialized pointer –**An uninitialized pointer stores an undefined value. A null pointer stores a defined value, but one that is defined by the environment to not be a valid address for any member or object.
2. **NULL vs Void Pointer** – Null pointer is a value, while void pointer is a type

## [**Wild pointer**](https://www.geeksforgeeks.org/what-are-wild-pointers-how-can-we-avoid/)

A pointer which has not been initialized to anything (not even NULL) is known as wild pointer. The pointer may be initialized to a non-NULL garbage value that may not be a valid address.

|  |
| --- |
| int main()  {      int \*p;  /\* wild pointer \*/      int x = 10;      // p is not a wild pointer now      p = &x;      return 0;  } |

# Difference between Structure and Union in C

## [**structures in C**](https://www.geeksforgeeks.org/structures-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.  
**Defining a structure:** To define a structure, you must use the **struct** statement. The struct statement defines a new data type, with more than one member. The format of the struct statement is as follows:

struct [structure name]

{

member definition;

member definition;

...

member definition;

};

## [**union**](https://www.geeksforgeeks.org/union-c/)

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.  
**Defining a Union:** To define a union, you must use the **union** statement in the same way as you did while defining a structure. The union statement defines a new data type with more than one member for your program. The format of the union statement is as follows:

union [union name]

{

member definition;

member definition;

...

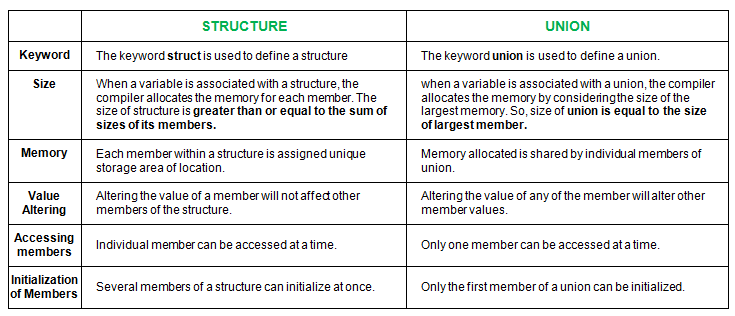
member definition;

};

## **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. **‘.’** operator is used for accessing members.

## **Differences**



|  |
| --- |
| // C program to illustrate differences  // between structure and Union  #include <stdio.h>  #include <string.h>    // declaring structure  struct struct\_example  {      int integer;      float decimal;      char name[20];  };    // declaraing union    union union\_example  {      int integer;      float decimal;      char name[20];  };    void main()  {      // creating variable for structure      // and initializing values difference      // six      struct struct\_example s={18,38,"geeksforgeeks"};        // creating variable for union      // and initializing values      union union\_example u={18,38,"geeksforgeeks"};          printf("structure data:\n integer: %d\n"                  "decimal: %.2f\n name: %s\n",                  s.integer, s.decimal, s.name);      printf("\nunion data:\n integeer: %d\n"                   "decimal: %.2f\n name: %s\n",                  u.integer, u.decimal, u.name);          // difference two and three      printf("\nsizeof structure : %d\n", sizeof(s));      printf("sizeof union : %d\n", sizeof(u));        // difference five      printf("\n Accessing all members at a time:");      s.integer = 183;      s.decimal = 90;      strcpy(s.name, "geeksforgeeks");        printf("structure data:\n integer: %d\n "                  "decimal: %.2f\n name: %s\n",              s.integer, s.decimal, s.name);        u.integer = 183;      u.decimal = 90;      strcpy(u.name, "geeksforgeeks");        printf("\nunion data:\n integeer: %d\n "                  "decimal: %.2f\n name: %s\n",              u.integer, u.decimal, u.name);        printf("\n Accessing one member at time:");        printf("\nstructure data:");      s.integer = 240;      printf("\ninteger: %d", s.integer);        s.decimal = 120;      printf("\ndecimal: %f", s.decimal);        strcpy(s.name, "C programming");      printf("\nname: %s\n", s.name);        printf("\n union data:");      u.integer = 240;      printf("\ninteger: %d", u.integer);        u.decimal = 120;      printf("\ndecimal: %f", u.decimal);        strcpy(u.name, "C programming");      printf("\nname: %s\n", u.name);        //difference four      printf("\nAltering a member value:\n");      s.integer = 1218;      printf("structure data:\n integer: %d\n "                  " decimal: %.2f\n name: %s\n",                  s.integer, s.decimal, s.name);        u.integer = 1218;      printf("union data:\n integer: %d\n"             " decimal: %.2f\n name: %s\n",              u.integer, u.decimal, u.name);  } |

Copy CodeRun on IDE

Output:

structure data:

integer: 18

decimal: 38.00

name: geeksforgeeks

union data:

integeer: 18

decimal: 0.00

name: ?

sizeof structure: 28

sizeof union: 20

Accessing all members at a time: structure data:

integer: 183

decimal: 90.00

name: geeksforgeeks

union data:

integeer: 1801807207

decimal: 277322871721159510000000000.00

name: geeksforgeeks

Accessing one member at a time:

structure data:

integer: 240

decimal: 120.000000

name: C programming

union data:

integer: 240

decimal: 120.000000

name: C programming

Altering a member value:

structure data:

integer: 1218

decimal: 120.00

name: C programming

union data:

integer: 1218

decimal: 0.00

name: ?

# Function Pointer in C

In C, like [normal data pointers](https://www.geeksforgeeks.org/pointers-in-c-and-c-set-1-introduction-arithmetic-and-array/)(int \*, char \*, etc), we can have pointers to functions. Following is a simple example that shows declaration and function call using function pointer.

|  |
| --- |
| #include <stdio.h>  // A normal function with an int parameter  // and void return type  void fun(int a)  {      printf("Value of a is %d\n", a);  }    int main()  {      // fun\_ptr is a pointer to function fun()      void (\*fun\_ptr)(int) = &fun;        /\* The above line is equivalent of following two         void (\*fun\_ptr)(int);         fun\_ptr = &fun;      \*/        // Invoking fun() using fun\_ptr      (\*fun\_ptr)(10);        return 0;  } |

Copy CodeRun on IDE

Output:

Value of a is 10

Why do we need an extra bracket around function pointers like fun\_ptr in above example?  
If we remove bracket, then the expression “void (\*fun\_ptr)(int)” becomes “void \*fun\_ptr(int)” which is declaration of a function that returns void pointer. See following post for details.  
[How to declare a pointer to a function?](https://www.geeksforgeeks.org/how-to-declare-a-pointer-to-a-function/)

## **Following are some interesting facts about function pointers.**

**1)** Unlike normal pointers, a function pointer points to code, not data. Typically a function pointer stores the start of executable code.

**2)**Unlike normal pointers, we do not allocate de-allocate memory using function pointers.

**3)** A function’s name can also be used to get functions’ address. For example, in the below program, we have removed address operator ‘&’ in assignment. We have also changed function call by removing \*, the program still works.

|  |
| --- |
| #include <stdio.h>  // A normal function with an int parameter  // and void return type  void fun(int a)  {      printf("Value of a is %d\n", a);  }    int main()  {      void (\*fun\_ptr)(int) = fun;  // & removed        fun\_ptr(10);  // \* removed        return 0;  } |

Copy CodeRun on IDE

Output:

Value of a is 10

**4)** Like normal pointers, we can have an array of function pointers. Below example in point 5 shows syntax for array of pointers.

**5)** Function pointer can be used in place of switch case. For example, in below program, user is asked for a choice between 0 and 2 to do different tasks.

|  |
| --- |
| #include <stdio.h>  void add(int a, int b)  {      printf("Addition is %d\n", a+b);  }  void subtract(int a, int b)  {      printf("Subtraction is %d\n", a-b);  }  void multiply(int a, int b)  {      printf("Multiplication is %d\n", a\*b);  }    int main()  {      // fun\_ptr\_arr is an array of function pointers      void (\*fun\_ptr\_arr[])(int, int) = {add, subtract, multiply};      unsigned int ch, a = 15, b = 10;        printf("Enter Choice: 0 for add, 1 for subtract and 2 "              "for multiply\n");      scanf("%d", &ch);        if (ch > 2) return 0;        (\*fun\_ptr\_arr[ch])(a, b);        return 0;  } |

Copy CodeRun on IDE

Enter Choice: 0 for add, 1 for subtract and 2 for multiply

2

Multiplication is 150

**6)**Like normal data pointers, a function pointer can be passed as an argument and can also be returned from a function.  
For example, consider the following C program where wrapper() receives a void fun() as parameter and calls the passed function.

|  |
| --- |
| // A simple C program to show function pointers as parameter  #include <stdio.h>    // Two simple functions  void fun1() { printf("Fun1\n"); }  void fun2() { printf("Fun2\n"); }    // A function that receives a simple function  // as parameter and calls the function  void wrapper(void (\*fun)())  {      fun();  }    int main()  {      wrapper(fun1);      wrapper(fun2);      return 0;  } |

Copy CodeRun on IDE

This point in particular is very useful in C. In C, we can use function pointers to avoid code redundancy. For example a simple [qsort()](http://www.cplusplus.com/reference/cstdlib/qsort/) function can be used to sort arrays in ascending order or descending or by any other order in case of array of structures. Not only this, with function pointers and void pointers, it is possible to use qsort for any data type.

|  |
| --- |
| // An example for qsort and comparator  #include <stdio.h>  #include <stdlib.h>    // A sample comparator function that is used  // for sorting an integer array in ascending order.  // To sort any array for any other data type and/or  // criteria, all we need to do is write more compare  // functions.  And we can use the same qsort()  int compare (const void \* a, const void \* b)  {    return ( \*(int\*)a - \*(int\*)b );  }    int main ()  {    int arr[] = {10, 5, 15, 12, 90, 80};    int n = sizeof(arr)/sizeof(arr[0]), i;      qsort (arr, n, sizeof(int), compare);      for (i=0; i<n; i++)       printf ("%d ", arr[i]);    return 0;  } |

Copy CodeRun on IDE

Output:

5 10 12 15 80 90

Similar to qsort(), we can write our own functions that can be used for any data type and can do different tasks without code redundancy. Below is an example search function that can be used for any data type. In fact we can use this search function to find close elements (below a threshold) by writing a customized compare function.

|  |
| --- |
| #include <stdio.h>  #include <stdbool.h>    // A compare function that is used for searching an integer  // array  bool compare (const void \* a, const void \* b)  {    return ( \*(int\*)a == \*(int\*)b );  }    // General purpose search() function that can be used  // for searching an element \*x in an array arr[] of  // arr\_size. Note that void pointers are used so that  // the function can be called by passing a pointer of  // any type.  ele\_size is size of an array element  int search(void \*arr, int arr\_size, int ele\_size, void \*x,             bool compare (const void \* , const void \*))  {      // Since char takes one byte, we can use char pointer      // for any type/ To get pointer arithmetic correct,      // we need to multiply index with size of an array      // element ele\_size      char \*ptr = (char \*)arr;        int i;      for (i=0; i<arr\_size; i++)          if (compare(ptr + i\*ele\_size, x))             return i;        // If element not found      return -1;  }    int main()  {      int arr[] = {2, 5, 7, 90, 70};      int n = sizeof(arr)/sizeof(arr[0]);      int x = 7;      printf ("Returned index is %d ", search(arr, n,                                 sizeof(int), &x, compare));      return 0;  } |

Copy CodeRun on IDE

Output:

Returned index is 2

The above search function can be used for any data type by writing a separate customized compare().

**7)** Many object oriented features in C++ are implemented using function pointers in C. For example [virtual functions](https://www.geeksforgeeks.org/virtual-functions-and-runtime-polymorphism-in-c-set-1-introduction/). Class methods are another example implemented using function pointers. Refer [this book](http://www.cs.rit.edu/~ats/books/ooc.pdf) for more details.

# 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 (current directory).  
**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 |

Run on IDE

**3)** The macros can take function like arguments, the arguments are not checked for data type. For example, the following macro INCREMENT(x) can be used for x of any data type.

|  |
| --- |
| #include <stdio.h>  #define INCREMENT(x) ++x  int main()  {      char \*ptr = "GeeksQuiz";      int x = 10;      printf("%s  ", INCREMENT(ptr));      printf("%d", INCREMENT(x));      return 0;  }  // Output: eeksQuiz 11 |

Run on IDE

**4)** The macro arguments are not evaluated before macro expansion. For example consider the following program

|  |
| --- |
| #include <stdio.h>  #define MULTIPLY(a, b) a\*b  int main()  {      // The macro is expended as 2 + 3 \* 3 + 5, not as 5\*8      printf("%d", MULTIPLY(2+3, 3+5));      return 0;  }  // Output: 16 |

Run on IDE

The previous problem can be solved using following program

|  |
| --- |
| #include <stdio.h>  //here, instead of writing a\*a we write (a)\*(b)  #define MULTIPLY(a, b) (a)\*(b)  int main()  {      // The macro is expended as (2 + 3) \* (3 + 5), as 5\*8      printf("%d", MULTIPLY(2+3, 3+5));      return 0;  }  //This code is contributed by Santanu  // Output: 40 |

Run on IDE

**5)** The tokens passed to macros can be concatenated using operator **##** called Token-Pasting operator.

|  |
| --- |
| #include <stdio.h>  #define merge(a, b) a##b  int main()  {      printf("%d ", merge(12, 34));  }  // Output: 1234 |

Run on IDE

**6)** A token passed to macro can be converted to a string literal by using # before it.

|  |
| --- |
| #include <stdio.h>  #define get(a) #a  int main()  {      // GeeksQuiz is changed to "GeeksQuiz"      printf("%s", get(GeeksQuiz));  }  // Output: GeeksQuiz |

Run on IDE

**7)** The macros can be written in multiple lines using ‘\’. The last line doesn’t need to have ‘\’.

|  |
| --- |
| #include <stdio.h>  #define PRINT(i, limit) while (i < limit) \                          { \                              printf("GeeksQuiz "); \                              i++; \                          }  int main()  {      int i = 0;      PRINT(i, 3);      return 0;  }  // Output: GeeksQuiz  GeeksQuiz  GeeksQuiz |

Run on IDE

**8)**The macros with arguments should be avoided as they cause problems sometimes. And Inline functions should be preferred as there is type checking parameter evaluation in inline functions. From [C99](http://en.wikipedia.org/wiki/C99) onward, inline functions are supported by C language also.  
For example consider the following program. From first look the output seems to be 1, but it produces 36 as output.

|  |
| --- |
| #define square(x) x\*x  int main()  {    int x = 36/square(6); // Expended as 36/6\*6    printf("%d", x);    return 0;  }  // Output: 36 |

Run on IDE

If we use inline functions, we get the expected output. Also the program given in point 4 above can be corrected using inline functions.

|  |
| --- |
| inline int square(int x) { return x\*x; }  int main()  {    int x = 36/square(6);    printf("%d", x);    return 0;  }  // Output: 1 |

Run on IDE

**9)** Preprocessors also support if-else directives which are typically used for conditional compilation.

|  |
| --- |
| int main()  {  #if VERBOSE >= 2    printf("Trace Message");  #endif  } |

Run on IDE

**10)** A header file may be included more than one time directly or indirectly, this leads to problems of redeclaration of same variables/functions. To avoid this problem, directives like ***defined***, ***ifdef***and ***ifndef***are used.  
**11)** There are some standard macros which can be used to print program file (\_\_FILE\_\_), Date of compilation (\_\_DATE\_\_), Time of compilation (\_\_TIME\_\_) and Line Number in C code (\_\_LINE\_\_)

|  |
| --- |
| #include <stdio.h>    int main()  {     printf("Current File :%s\n", \_\_FILE\_\_ );     printf("Current Date :%s\n", \_\_DATE\_\_ );     printf("Current Time :%s\n", \_\_TIME\_\_ );     printf("Line Number :%d\n", \_\_LINE\_\_ );     return 0;  }    /\* Output:  Current File :C:\Users\GfG\Downloads\deleteBST.c  Current Date :Feb 15 2014  Current Time :07:04:25  Line Number :8 \*/ |

Run on IDE

**12)** We can remove already defined macros using:   
**#undef MACRO\_NAME**

|  |
| --- |
| #include <stdio.h>  #define LIMIT 100  int main()  {     printf("%d",LIMIT);     //removing defined macro LIMIT     #undef LIMIT     //Next line causes error as LIMIT is not defined     printf("%d",LIMIT);     return 0;  }  //This code is contributed by Santanu |

**Run on IDE**

**Following program is executed correctly as we have declare LIMIT as an integer variable after removing previously defined macro LIMIT**

|  |
| --- |
| #include <stdio.h>  #define LIMIT 1000  int main()  {     printf("%d",LIMIT);     //removing defined macro LIMIT     #undef LIMIT     //Declare LIMIT as integer again     int LIMIT=1001;     printf("\n%d",LIMIT);     return 0;  }  //This code is contributed by Santanu    /\*Output is :  1000  1001  \*/ |

**Run on IDE**

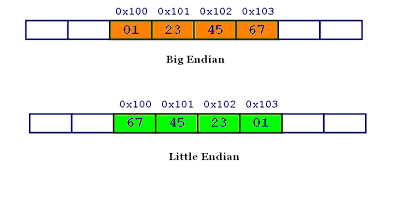
**Another interesting fact about macro using (#undef)**

|  |
| --- |
| #include <stdio.h>  //div function prototype  float div(float, float);  #define div(x, y) x/y    int main()  {  //use of macro div  //Note: %0.2f for taking two decimal value after point  printf("%0.2f",div(10.0,5.0));  //removing defined macro div  #undef div  //function div is called as macro definition is removed  printf("\n%0.2f",div(10.0,5.0));  return 0;  }    //div function definition  float div(float x, float y){  return y/x;  }  //This code is contributed by Santanu    /\*Output is :  2.00  0.50  \*/ |

Little and Big Endian Mystery

## What are these?

Little and big endian are two ways of storing multibyte data-types ( int, float, etc). In little endian machines, last byte of binary representation of the multibyte data-type is stored first. On the other hand, in big endian machines, first byte of binary representation of the multibyte data-type is stored first.  
  
Suppose integer is stored as 4 bytes (For those who are using DOS based compilers such as C++ 3.0 , integer is 2 bytes) then a variable x with value 0x01234567 will be stored as following.

[](http://4.bp.blogspot.com/_IEmaCFe3y9g/SO3GGEF4UkI/AAAAAAAAAAc/z7waF2Lwg0s/s1600-h/lb.GIF)

Memory representation of integer ox01234567 inside Big and little endian machines

**How to see memory representation of multibyte data types on your machine?**  
Here is a sample C code that shows the byte representation of int, float and pointer.

|  |
| --- |
| #include <stdio.h>    /\* function to show bytes in memory, from location start to start+n\*/  void show\_mem\_rep(char \*start, int n)  {      int i;      for (i = 0; i < n; i++)           printf(" %.2x", start[i]);      printf("\n");  }    /\*Main function to call above function for 0x01234567\*/  int main()  {     int i = 0x01234567;     show\_mem\_rep((char \*)&i, sizeof(i));     getchar();     return 0;  } |

Copy CodeRun on IDE

When above program is run on little endian machine, gives “67 45 23 01” as output , while if it is run on endian machine, gives “01 23 45 67” as output.

**Is there a quick way to determine endianness of your machine?**  
There are n no. of ways for determining endianness of your machine. Here is one quick way of doing the same.

|  |
| --- |
| #include <stdio.h>  int main()  {     unsigned int i = 1;     char \*c = (char\*)&i;     if (\*c)         printf("Little endian");     else         printf("Big endian");     getchar();     return 0;  } |

Copy CodeRun on IDE

In the above program, a character pointer c is pointing to an integer i. Since size of character is 1 byte when the character pointer is de-referenced it will contain only first byte of integer. If machine is little endian then \*c will be 1 (because last byte is stored first) and if machine is big endian then \*c will be 0.  
 **Does endianness matter for programmers?**  
Most of the times compiler takes care of endianness, however, endianness becomes an issue in following cases.

It matters in network programming: Suppose you write integers to file on a little endian machine and you transfer this file to a big endian machine. Unless there is little endian to big endian transformation, big endian machine will read the file in reverse order. You can find such a practical example here.

Standard byte order for networks is big endian, also known as network byte order. Before transferring data on network, data is first converted to network byte order (big endian).

Sometimes it matters when you are using type casting, below program is an example.

|  |
| --- |
| #include <stdio.h>  int main()  {      unsigned char arr[2] = {0x01, 0x00};      unsigned short int x = \*(unsigned short int \*) arr;      printf("%d", x);      getchar();      return 0;  } |

Copy CodeRun on IDE

In the above program, a char array is typecasted to an unsigned short integer type. When I run above program on little endian machine, I get 1 as output, while if I run it on a big endian machine I get 256. To make programs endianness independent, above programming style should be avoided.  
 **What are bi-endians?**  
Bi-endian processors can run in both modes little and big endian.

**What are the examples of little, big endian and bi-endian machines ?**  
Intel based processors are little endians. ARM processors were little endians. Current generation ARM processors are bi-endian.

Motorola 68K processors are big endians. PowerPC (by Motorola) and SPARK (by Sun) processors were big endian. Current version of these processors are bi-endians.  
 **Does endianness affects file formats?**  
File formats which have 1 byte as a basic unit are independent of endianness e.g., ASCII files . Other file formats use some fixed endianness forrmat e.g, JPEG files are stored in big endian format.

**Which one is better — little endian or big endian?**  
The term little and big endian came from Gulliver’s Travels by Jonathan Swift. Two groups could not agree by which end an egg should be opened -a-the little or the big. Just like the egg issue, there is no technological reason to choose one byte ordering convention over the other, hence the arguments degenerate into bickering about sociopolitical issues. As long as one of the conventions is selected and adhered to consistently, the choice is arbitrary.

# Macros vs Functions

Macros are **pre-processed** which means that all the macros would be processed before your program compiles. However, functions are **not preprocessed but compiled**.

**See the following example of Macro:**

|  |
| --- |
| #include<stdio.h>  #define NUMBER 10  int main()  {       printf("%d", NUMBER);       return 0;  } |

Copy CodeRun on IDE

**Output:**

10

**See the following example of Function:**

|  |
| --- |
| #include<stdio.h>  int number()  {      return 10;  }  int main()  {      printf("%d", number());      return 0;  } |

Copy CodeRun on IDE

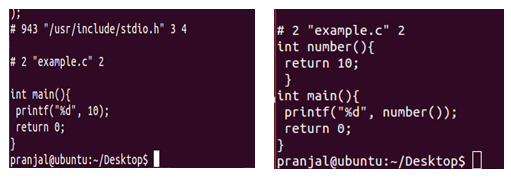
**Output:**

10

Now compile them using the command:

**gcc –E file\_name.c**

This will give you the executable code as shown in the figure:

[](https://www.geeksforgeeks.org/wp-content/uploads/b12.png)

This shows that the macros are preprocessed while functions are not.

In macros, no type checking(incompatible operand, etc.) is done and thus use of micros can lead to errors/side-effects in some cases. However, this is not the case with functions. Also, macros do not check for compilation error (if any). Consider the following two codes:

**Macros:**

|  |
| --- |
| #include<stdio.h>  #define CUBE(b) b\*b\*b  int main()  {       printf("%d", CUBE(1+2));       return 0;  } |

Copy CodeRun on IDE

**Output: Unexpected output**

7

**Functions:**

|  |
| --- |
| #include<stdio.h>  int cube(int a)  {       return a\*a\*a;  }  int main()  {      printf("%d", cube(1+2));      return 0;  } |

Copy CodeRun on IDE

**Output: As expected**

27

* Macros are usually one liner. However, they can consist of more than one line, Click [here](https://www.geeksforgeeks.org/multiline-macros-in-c/) to see the usage. There are no such constraints in functions.
* The speed at which macros and functions differs. Macros are typically faster than functions as they don’t involve actual function call overhead.

**Conclusion:**  
Macros are no longer recommended as they cause following issues. There is a better way in modern compilers that is inline functions and const variable. Below are disadvantages of macros:

a) There is no type checking

b) Difficult to debug as they cause simple replacement.

c) Macro don’t have namespace, so a macro in one section of code can affect other section.

d) Macros can cause side effects as shown in above CUBE() example.

Memory Layout of C Programs

A typical memory representation of C program consists of following sections.

1. Text segment  
2. Initialized data segment  
3. Uninitialized data segment  
4. Stack  
5. Heap

[](https://cdncontribute.geeksforgeeks.org/wp-content/uploads/memoryLayoutC.jpg)  
A typical memory layout of a running process

**1. Text Segment:**  
A text segment , also known as a code segment or simply as text, is one of the sections of a program in an object file or in memory, which contains executable instructions.

As a memory region, a text segment may be placed below the heap or stack in order to prevent heaps and stack overflows from overwriting it.

Usually, the text segment is sharable so that only a single copy needs to be in memory for frequently executed programs, such as text editors, the C compiler, the shells, and so on. Also, the text segment is often read-only, to prevent a program from accidentally modifying its instructions.

**2. Initialized Data Segment:**  
Initialized data segment, usually called simply the Data Segment. A data segment is a portion of virtual address space of a program, which contains the global variables and static variables that are initialized by the programmer.

Note that, data segment is not read-only, since the values of the variables can be altered at run time.

This segment can be further classified into initialized read-only area and initialized read-write area.

For instance the global string defined by char s[] = “hello world” in C and a C statement like int debug=1 outside the main (i.e. global) would be stored in initialized read-write area. And a global C statement like const char\* string = “hello world” makes the string literal “hello world” to be stored in initialized read-only area and the character pointer variable string in initialized read-write area.

Ex: static int i = 10 will be stored in data segment and global int i = 10 will also be stored in data segment

**3. Uninitialized Data Segment:**  
Uninitialized data segment, often called the “bss” segment, named after an ancient assembler operator that stood for “block started by symbol.” Data in this segment is initialized by the kernel to arithmetic 0 before the program starts executing

uninitialized data starts at the end of the data segment and contains all global variables and static variables that are initialized to zero or do not have explicit initialization in source code.

For instance a variable declared static int i; would be contained in the BSS segment.  
For instance a global variable declared int j; would be contained in the BSS segment.

**4. Stack:**  
The stack area traditionally adjoined the heap area and grew the opposite direction; when the stack pointer met the heap pointer, free memory was exhausted. (With modern large address spaces and virtual memory techniques they may be placed almost anywhere, but they still typically grow opposite directions.)

The stack area contains the program stack, a LIFO structure, typically located in the higher parts of memory. On the standard PC x86 computer architecture it grows toward address zero; on some other architectures it grows the opposite direction. A “stack pointer” register tracks the top of the stack; it is adjusted each time a value is “pushed” onto the stack. The set of values pushed for one function call is termed a “stack frame”; A stack frame consists at minimum of a return address.

Stack, where automatic variables are stored, along with information that is saved each time a function is called. Each time a function is called, the address of where to return to and certain information about the caller’s environment, such as some of the machine registers, are saved on the stack. The newly called function then allocates room on the stack for its automatic and temporary variables. This is how recursive functions in C can work. Each time a recursive function calls itself, a new stack frame is used, so one set of variables doesn’t interfere with the variables from another instance of the function.

**5. Heap:**  
Heap is the segment where dynamic memory allocation usually takes place.

The heap area begins at the end of the BSS segment and grows to larger addresses from there.The Heap area is managed by malloc, realloc, and free, which may use the brk and sbrk system calls to adjust its size (note that the use of brk/sbrk and a single “heap area” is not required to fulfill the contract of malloc/realloc/free; they may also be implemented using mmap to reserve potentially non-contiguous regions of virtual memory into the process’ virtual address space). The Heap area is shared by all shared libraries and dynamically loaded modules in a process.

Examples.

The size(1) command reports the sizes (in bytes) of the text, data, and bss segments. ( for more details please refer man page of size(1) )

1. Check the following simple C program

|  |
| --- |
| #include <stdio.h>    int main(void)  {      return 0;  } |

Copy CodeRun on IDE

[narendra@CentOS]$ gcc memory-layout.c -o memory-layout

[narendra@CentOS]$ size memory-layout

text data bss dec hex filename

960 248 8 1216 4c0 memory-layout

2. Let us add one global variable in program, now check the size of bss (highlighted in red color).

|  |
| --- |
| #include <stdio.h>    int global; /\* Uninitialized variable stored in bss\*/    int main(void)  {      return 0;  } |

Copy CodeRun on IDE

[narendra@CentOS]$ gcc memory-layout.c -o memory-layout

[narendra@CentOS]$ size memory-layout

text data bss dec hex filename

960 248 **12** 1220 4c4 memory-layout

3. Let us add one static variable which is also stored in bss.

|  |
| --- |
| #include <stdio.h>    int global; /\* Uninitialized variable stored in bss\*/    int main(void)  {      static int i; /\* Uninitialized static variable stored in bss \*/      return 0;  } |

Copy CodeRun on IDE

[narendra@CentOS]$ gcc memory-layout.c -o memory-layout

[narendra@CentOS]$ size memory-layout

text data bss dec hex filename

960 248 **16** 1224 4c8 memory-layout

4. Let us initialize the static variable which will then be stored in Data Segment (DS)

|  |
| --- |
| #include <stdio.h>    int global; /\* Uninitialized variable stored in bss\*/    int main(void)  {      static int i = 100; /\* Initialized static variable stored in DS\*/      return 0;  } |

Copy CodeRun on IDE

[narendra@CentOS]$ gcc memory-layout.c -o memory-layout

[narendra@CentOS]$ size memory-layout

text data bss dec hex filename

960 **252 12** 1224 4c8 memory-layout

5. Let us initialize the global variable which will then be stored in Data Segment (DS)

|  |
| --- |
| #include <stdio.h>    int global = 10; /\* initialized global variable stored in DS\*/    int main(void)  {      static int i = 100; /\* Initialized static variable stored in DS\*/      return 0;  } |

Copy CodeRun on IDE

[narendra@CentOS]$ gcc memory-layout.c -o memory-layout

[narendra@CentOS]$ size memory-layout

text data bss dec hex filename

960 **256 8** 1224 4c8 memory-layout

# Storage Classes in C

Storage Classes are used to describe about the features of a variable/function. These features basically include the scope, visibility and life-time which help us to trace the existence of a particular variable during the runtime of a program.

C language uses 4 storage classes, namely:

## **auto**

This is the default storage class for all the variables declared inside a function or a block. Hence, the keyword auto is rarely used while writing programs in C language. Auto variables can be only accessed within the block/function they have been declared and not outside them (which defines their scope). Of course, these can be accessed within nested blocks within the parent block/function in which the auto variable was declared. However, they can be accessed outside their scope as well using the concept of pointers given here by pointing to the very exact memory location where the variables resides. They are assigned a garbage value by default whenever they are declared.

## [**extern**](https://www.geeksforgeeks.org/understanding-extern-keyword-in-c/)

Extern storage class simply tells us that the variable is defined elsewhere and not within the same block where it is used. Basically, the value is assigned to it in a different block and this can be overwritten/changed in a different block as well. So an extern variable is nothing but a global variable initialized with a legal value where it is declared in order to be used elsewhere. It can be accessed within any function/block. Also, a normal global variable can be made extern as well by placing the ‘extern’ keyword before its declaration/definition in any function/block. This basically signifies that we are not initializing a new variable but instead we are using/accessing the global variable only. The main purpose of using extern variables is that they can be accessed between two different files which are part of a large program. For more information on how extern variables work, have a look at this [link](https://www.geeksforgeeks.org/understanding-extern-keyword-in-c/).

## [**static**](http://quiz.geeksforgeeks.org/static-variables-in-c/)

This storage class is used to declare static variables which are popularly used while writing programs in C language. Static variables have a property of preserving their value even after they are out of their scope! Hence, static variables preserve the value of their last use in their scope. So we can say that they are initialized only once and exist till the termination of the program. Thus, no new memory is allocated because they are not re-declared. Their scope is local to the function to which they were defined. Global static variables can be accessed anywhere in the program. By default, they are assigned the value 0 by the compiler.

## [**register**](https://www.geeksforgeeks.org/understanding-register-keyword/)

This storage class declares register variables which have the same functionality as that of the auto variables. The only difference is that the compiler tries to store these variables in the register of the microprocessor if a free register is available. This makes the use of register variables to be much faster than that of the variables stored in the memory during the runtime of the program. If a free register is not available, these are then stored in the memory only. Usually few variables which are to be accessed very frequently in a program are declared with the register keyword which improves the running time of the program. An important and interesting point to be noted here is that we cannot obtain the address of a register variable using pointers.

To specify the storage class for a variable, the following syntax is to be followed:

Syntax:

storage\_class var\_data\_type var\_name;

Functions follow the same syntax as given above for variables. Have a look at the following C example for further clarification:

|  |
| --- |
| // A C program to demonstrate different storage  // classes  #include <stdio.h>    // declaring and initializing an extern variable  extern int x = 9;    // declaring and initialing a global variable z  // simply int z; would have initialized z with  // the default value of a global variable which is 0  int z = 10;    int main()  {      // declaring an auto variable (simply      // writing "int a=32;" works as well)      auto int a = 32;        // declaring a register variable      register char b = 'G';        // telling the compiler that the variable      // z is an extern variable and has been      // defined elsewhere (above the main      // function)      extern int z;        printf("Hello World!\n");        // printing the auto variable 'a'      printf("\nThis is the value of the auto "             " integer 'a': %d\n",a);        // printing the extern variables 'x'      // and 'z'      printf("\nThese are the values of the"             " extern integers 'x' and 'z'"             " respectively: %d and %d\n", x, z);        // printing the register variable 'b'      printf("\nThis is the value of the "             "register character 'b': %c\n",b);        // value of extern variable x modified      x = 2;        // value of extern variable z modified      z = 5;        // printing the modified values of      // extern variables 'x' and 'z'      printf("\nThese are the modified values "             "of the extern integers 'x' and "             "'z' respectively: %d and %d\n",x,z);        // using a static variable 'y'      printf("\n'y' is a static variable and its "             "value is NOT initialized to 5 after"             " the first iteration! See for"             " yourself :)\n");        while (x > 0)      {          static int y = 5;          y++;            // printing value of y at each iteration          printf("The value of y is %d\n",y);          x--;      }        // exiting      printf("\nBye! See you soon. :)\n");        return 0;  } |

Copy CodeRun on IDE

Output:

Hello World!

This is the value of the auto integer 'a': 32

These are the values of the extern integers 'x' and 'z'

respectively: 9 and 10

This is the value of the register character 'b': G

These are the modified values of the extern integers 'x'

and 'z' respectively: 2 and 5

'y' is a static variable and its value is NOT initialized

to 5 after the first iteration! See for yourself :)

The value of y is 6

The value of y is 7

Bye! See you soon. :)

# Structure Member Alignment, Padding and Data Packing

What do we mean by data alignment, structure packing and padding?

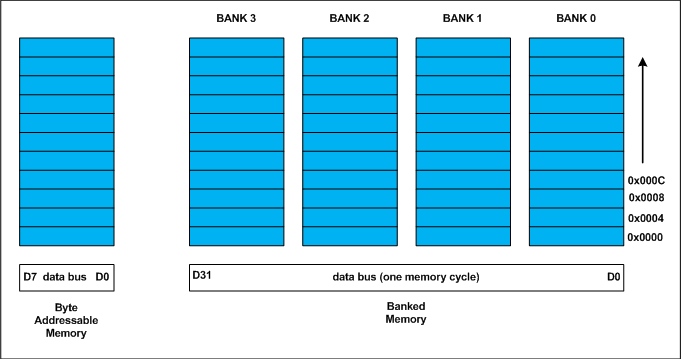
Predict the output of following program.

|  |
| --- |
| #include <stdio.h>    // Alignment requirements  // (typical 32 bit machine)    // char         1 byte  // short int    2 bytes  // int          4 bytes  // double       8 bytes    // structure A  typedef struct structa\_tag  {     char        c;     short int   s;  } structa\_t;    // structure B  typedef struct structb\_tag  {     short int   s;     char        c;     int         i;  } structb\_t;    // structure C  typedef struct structc\_tag  {     char        c;     double      d;     int         s;  } structc\_t;    // structure D  typedef struct structd\_tag  {     double      d;     int         s;     char        c;  } structd\_t;    int main()  {     printf("sizeof(structa\_t) = %d\n", sizeof(structa\_t));     printf("sizeof(structb\_t) = %d\n", sizeof(structb\_t));     printf("sizeof(structc\_t) = %d\n", sizeof(structc\_t));     printf("sizeof(structd\_t) = %d\n", sizeof(structd\_t));       return 0;  } |

Copy CodeRun on IDE

Before moving further, write down your answer on a paper, and read on. If you urge to see explanation, you may miss to understand any lacuna in your analogy. **Data Alignment:**

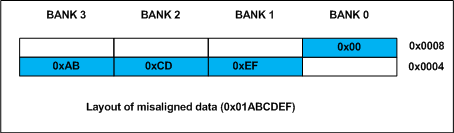
Every data type in C/C++ will have alignment requirement (infact it is mandated by processor architecture, not by language). A processor will have processing word length as that of data bus size. On a 32 bit machine, the processing word size will be 4 bytes.

[](https://www.geeksforgeeks.org/wp-content/uploads/MemoryAlignment1.gif)

Historically memory is byte addressable and arranged sequentially. If the memory is arranged as single bank of one byte width, the processor needs to issue 4 memory read cycles to fetch an integer. It is more economical to read all 4 bytes of integer in one memory cycle. To take such advantage, the memory will be arranged as group of 4 banks as shown in the above figure.

The memory addressing still be sequential. If bank 0 occupies an address X, bank 1, bank 2 and bank 3 will be at (X + 1), (X + 2) and (X + 3) addresses. If an integer of 4 bytes is allocated on X address (X is multiple of 4), the processor needs only one memory cycle to read entire integer.

Where as, if the integer is allocated at an address other than multiple of 4, it spans across two rows of the banks as shown in the below figure. Such an integer requires two memory read cycle to fetch the data.

[](https://www.geeksforgeeks.org/wp-content/uploads/MemoryAlignment2.gif)

A variable’s ***data alignment*** deals with the way the data stored in these banks. For example, the natural alignment of ***int*** on 32-bit machine is 4 bytes. When a data type is naturally aligned, the CPU fetches it in minimum read cycles.

Similarly, the natural alignment of ***short int*** is 2 bytes. It means, a ***short int*** can be stored in bank 0 – bank 1 pair or bank 2 – bank 3 pair. A ***double*** requires 8 bytes, and occupies two rows in the memory banks. Any misalignment of ***double*** will force more than two read cycles to fetch ***double*** data.

Note that a **double** variable will be allocated on 8 byte boundary on 32 bit machine and requires two memory read cycles. On a 64 bit machine, based on number of banks, **double** variable will be allocated on 8 byte boundary and requires only one memory read cycle.

**Structure Padding:**

In C/C++ a structures are used as data pack. It doesn’t provide any data encapsulation or data hiding features (C++ case is an exception due to its semantic similarity with classes).

Because of the alignment requirements of various data types, every member of structure should be naturally aligned. The members of structure allocated sequentially increasing order. Let us analyze each struct declared in the above program.

**Output of Above Program:**

**For the sake of convenience, assume every structure type variable is allocated on 4 byte boundary (say 0x0000), i.e. the base address of structure is multiple of 4 (need not necessary always, see explanation of structc\_t).**

**structure A**

The *structa\_t* first element is *char* which is one byte aligned, followed by *short int*. short int is 2 byte aligned. If the the short int element is immediately allocated after the char element, it will start at an odd address boundary. The compiler will insert a padding byte after the char to ensure short int will have an address multiple of 2 (i.e. 2 byte aligned). The total size of structa\_t will be sizeof(char) + 1 (padding) + sizeof(short), 1 + 1 + 2 = 4 bytes.

**structure B**

The first member of *structb\_t* is short int followed by char. Since char can be on any byte boundary no padding required in between short int and char, on total they occupy 3 bytes. The next member is int. If the int is allocated immediately, it will start at an odd byte boundary. We need 1 byte padding after the char member to make the address of next int member is 4 byte aligned. On total, the *structb\_t* requires 2 + 1 + 1 (padding) + 4 = 8 bytes.

**structure C – Every structure will also have alignment requirements**

Applying same analysis, *structc\_t* needs sizeof(char) + 7 byte padding + sizeof(double) + sizeof(int) = 1 + 7 + 8 + 4 = 20 bytes. However, the sizeof(structc\_t) will be 24 bytes. It is because, along with structure members, structure type variables will also have natural alignment. Let us understand it by an example. Say, we declared an array of structc\_t as shown below

structc\_t structc\_array[3];

Assume, the base address of *structc\_array* is 0x0000 for easy calculations. If the structc\_t occupies 20 (0x14) bytes as we calculated, the second structc\_t array element (indexed at 1) will be at 0x0000 + 0x0014 = 0x0014. It is the start address of index 1 element of array. The double member of this structc\_t will be allocated on 0x0014 + 0x1 + 0x7 = 0x001C (decimal 28) which is not multiple of 8 and conflicting with the alignment requirements of double. As we mentioned on the top, the alignment requirement of double is 8 bytes.

Inorder to avoid such misalignment, compiler will introduce alignment requirement to every structure. It will be as that of the largest member of the structure. In our case alignment of structa\_t is 2, structb\_t is 4 and structc\_t is 8. If we need nested structures, the size of largest inner structure will be the alignment of immediate larger structure.

In structc\_t of the above program, there will be padding of 4 bytes after int member to make the structure size multiple of its alignment. Thus the sizeof (structc\_t) is 24 bytes. It guarantees correct alignment even in arrays. You can cross check.

**structure D – How to Reduce Padding?**

By now, it may be clear that padding is unavoidable. There is a way to minimize padding. The programmer should declare the structure members in their increasing/decreasing order of size. An example is structd\_t given in our code, whose size is 16 bytes in lieu of 24 bytes of structc\_t.

**What is structure packing?**

Some times it is mandatory to avoid padded bytes among the members of structure. For example, reading contents of ELF file header or BMP or JPEG file header. We need to define a structure similar to that of the header layout and map it. However, care should be exercised in accessing such members. Typically reading byte by byte is an option to avoid misaligned exceptions. There will be hit on performance.

Most of the compilers provide non standard extensions to switch off the default padding like pragmas or command line switches. Consult the documentation of respective compiler for more details.

**Pointer Mishaps:**

There is possibility of potential error while dealing with pointer arithmetic. For example, dereferencing a generic pointer (void \*) as shown below can cause misaligned exception,

// Deferencing a generic pointer (not safe)

// There is no guarantee that pGeneric is integer aligned

\*(int \*)pGeneric;

It is possible above type of code in programming. If the pointer *pGeneric* is not aligned as per the requirements of casted data type, there is possibility to get misaligned exception.

Infact few processors will not have the last two bits of address decoding, and there is no way to access *misaligned* address. The processor generates misaligned exception, if the programmer tries to access such address.

**A note on malloc() returned pointer**

The pointer returned by malloc() is *void \**. It can be converted to any data type as per the need of programmer. The implementer of malloc() should return a pointer that is aligned to maximum size of primitive data types (those defined by compiler). It is usually aligned to 8 byte boundary on 32 bit machines.

**Object File Alignment, Section Alignment, Page Alignment**

These are specific to operating system implementer, compiler writers and are beyond the scope of this article. Infact, I don’t have much information.

**General Questions:**

1. Is alignment applied for stack?

Yes. The stack is also memory. The system programmer should load the stack pointer with a memory address that is properly aligned. Generally, the processor won’t check stack alignment, it is the programmer’s responsibility to ensure proper alignment of stack memory. Any misalignment will cause run time surprises.

For example, if the processor word length is 32 bit, stack pointer also should be aligned to be multiple of 4 bytes.

2. If *char* data is placed in a bank other bank 0, it will be placed on wrong data lines during memory read. How the processor handles *char* type?

Usually, the processor will recognize the data type based on instruction (e.g. LDRB on ARM processor). Depending on the bank it is stored, the processor shifts the byte onto least significant data lines.

3. When arguments passed on stack, are they subjected to alignment?

Yes. The compiler helps programmer in making proper alignment. For example, if a 16-bit value is pushed onto a 32-bit wide stack, the value is automatically padded with zeros out to 32 bits. Consider the following program.

|  |
| --- |
| void argument\_alignment\_check( char c1, char c2 )  {     // Considering downward stack     // (on upward stack the output will be negative)     printf("Displacement %d\n", (int)&c2 - (int)&c1);  } |

Copy CodeRun on IDE

The output will be 4 on a 32 bit machine. It is because each character occupies 4 bytes due to alignment requirements.

4. What will happen if we try to access a misaligned data?

It depends on processor architecture. If the access is misaligned, the processor automatically issues sufficient memory read cycles and packs the data properly onto the data bus. The penalty is on performance. Where as few processors will not have last two address lines, which means there is no-way to access odd byte boundary. Every data access must be aligned (4 bytes) properly. A misaligned access is critical exception on such processors. If the exception is ignored, read data will be incorrect and hence the results.

5. Is there any way to query alignment requirements of a data type.

Yes. Compilers provide non standard extensions for such needs. For example, \_\_alignof() in Visual Studio helps in getting the alignment requirements of data type. Read MSDN for details.

6. When memory reading is efficient in reading 4 bytes at a time on 32 bit machine, why should a **double** type be aligned on 8 byte boundary?

It is important to note that most of the processors will have math co-processor, called Floating Point Unit (FPU). Any floating point operation in the code will be translated into FPU instructions. The main processor is nothing to do with floating point execution. All this will be done behind the scenes.

As per standard, double type will occupy 8 bytes. And, every floating point operation performed in FPU will be of 64 bit length. Even float types will be promoted to 64 bit prior to execution.

The 64 bit length of FPU registers forces double type to be allocated on 8 byte boundary. I am assuming (I don’t have concrete information) in case of FPU operations, data fetch might be different, I mean the data bus, since it goes to FPU. Hence, the address decoding will be different for double types (which is expected to be on 8 byte boundary). It means, *the address decoding circuits of floating point unit will not have last 3 pins*.

**Answers:**

sizeof(structa\_t) = 4

sizeof(structb\_t) = 8

sizeof(structc\_t) = 24

sizeof(structd\_t) = 16

# Understanding “volatile” qualifier in C | Set 2 (Examples)

The volatile keyword is intended to prevent the compiler from applying any optimizations on objects that can change in ways that cannot be determined by the compiler.

Objects declared as volatile are omitted from optimization because their values can be changed by code outside the scope of current code at any time. The system always reads the current value of a volatile object from the memory location rather than keeping its value in temporary register at the point it is requested, even if a previous instruction asked for a value from the same object. So the simple question is, how can value of a variable change in such a way that compiler cannot predict. Consider the following cases for answer to this question.  
1) Global variables modified by an interrupt service routine outside the scope: For example, a global variable can represent a data port (usually global pointer referred as memory mapped IO) which will be updated dynamically. The code reading data port must be declared as volatile in order to fetch latest data available at the port. Failing to declare variable as volatile, the compiler will optimize the code in such a way that it will read the port only once and keeps using the same value in a temporary register to speed up the program (speed optimization). In general, an ISR used to update these data port when there is an interrupt due to availability of new data

2) Global variables within a multi-threaded application: There are multiple ways for threads communication, viz, message passing, shared memory, mail boxes, etc. A global variable is weak form of shared memory. When two threads sharing information via global variable, they need to be qualified with volatile. Since threads run asynchronously, any update of global variable due to one thread should be fetched freshly by another consumer thread. Compiler can read the global variable and can place them in temporary variable of current thread context. To nullify the effect of compiler optimizations, such global variables to be qualified as volatile

If we do not use volatile qualifier, the following problems may arise  
1) Code may not work as expected when optimization is turned on.  
2) Code may not work as expected when interrupts are enabled and used.

Let us see an example to understand how compilers interpret volatile keyword. Consider below code, we are changing value of const object using pointer and we are compiling code without optimization option. Hence compiler won’t do any optimization and will change value of const object.

|  |
| --- |
| /\* Compile code without optimization option \*/  #include <stdio.h>  int main(void)  {      const int local = 10;      int \*ptr = (int\*) &local;        printf("Initial value of local : %d \n", local);        \*ptr = 100;        printf("Modified value of local: %d \n", local);        return 0;  } |

Copy CodeRun on IDE

When we compile code with “–save-temps” option of gcc it generates 3 output files

1) preprocessed code (having .i extention)  
2) assembly code (having .s extention) and  
3) object code (having .o option).

We compile code without optimization, that’s why the size of assembly code will be larger (which is highlighted in red color below).

Output:

[narendra@ubuntu]$ gcc volatile.c -o volatile –save-temps

[narendra@ubuntu]$ ./volatile

Initial value of local : 10

Modified value of local: 100

[narendra@ubuntu]$ ls -l volatile.s

-rw-r–r– 1 narendra narendra 731 2016-11-19 16:19 volatile.s

[narendra@ubuntu]$

Let us compile same code with optimization option (i.e. -O option). In thr below code, “local” is declared as const (and non-volatile), GCC compiler does optimization and ignores the instructions which try to change value of const object. Hence value of const object remains same.

|  |
| --- |
| /\* Compile code with optimization option \*/  #include <stdio.h>    int main(void)  {      const int local = 10;      int \*ptr = (int\*) &local;        printf("Initial value of local : %d \n", local);        \*ptr = 100;        printf("Modified value of local: %d \n", local);        return 0;  } |

Copy CodeRun on IDE

For above code, compiler does optimization, that’s why the size of assembly code will reduce.

Output:

[narendra@ubuntu]$ gcc -O3 volatile.c -o volatile –save-temps

[narendra@ubuntu]$ ./volatile

Initial value of local : 10

Modified value of local: 10

[narendra@ubuntu]$ ls -l volatile.s

-rw-r–r– 1 narendra narendra 626 2016-11-19 16:21 volatile.s

Let us declare const object as volatile and compile code with optimization option. Although we compile code with optimization option, value of const object will change, because variable is declared as volatile that means don’t do any optimization.

|  |
| --- |
| /\* Compile code with optimization option \*/  #include <stdio.h>    int main(void)  {      const volatile int local = 10;      int \*ptr = (int\*) &local;        printf("Initial value of local : %d \n", local);        \*ptr = 100;        printf("Modified value of local: %d \n", local);        return 0;  } |

Copy CodeRun on IDE

Output:

[narendra@ubuntu]$ gcc -O3 volatile.c -o volatile –save-temp

[narendra@ubuntu]$ ./volatile

Initial value of local : 10

Modified value of local: 100

[narendra@ubuntu]$ ls -l volatile.s

-rw-r–r– 1 narendra narendra 711 2016-11-19 16:22 volatile.s

[narendra@ubuntu]$

The above example may not be a good practical example, the purpose was to explain how compilers interpret volatile keyword. As a practical example, think of touch sensor on mobile phones. The driver abstracting touch sensor will read the location of touch and send it to higher level applications. The driver itself should not modify (const-ness) the read location, and make sure it reads the touch input every time fresh (volatile-ness). Such driver must read the touch sensor input in const volatile manner.

# calloc() versus malloc()

The name **malloc** and calloc() are library functions that allocate memory dynamically. It means that memory is allocated during runtime(execution of the program) from heap segment.

* **Initialization:** malloc() allocates memory block of given size (in bytes) and returns a pointer to the beginning of the block. malloc() doesn’t initialize the allocated memory. If we try to acess the content of memory block then we’ll get garbage values.

|  |
| --- |
| void \* malloc( size\_t size ); |

* Copy Code
* calloc() allocates the memory and also initializes the allocates memory block to zero. If we try to access the content of these blocks then we’ll get 0.

|  |
| --- |
| void \* calloc( size\_t num, size\_t size ); |

* Copy Code

**Number of arguments:** Unlike malloc(), calloc() takes two arguments:  
1) Number of blocks to be allocated.  
2) Size of each block.

* **Return Value:** After successfull allocation in malloc() and calloc(), a pointer to the block of memory is returned otherwise **NULL** value is returned which indicates the failure of allocation.

For instance, If we want to allocate memory for array of 5 integers, see the following program:-

|  |
| --- |
| // C program to demonstrate the use of calloc()  // and malloc()  #include <stdio.h>  #include <stdlib.h>    int main()  {     int \*arr;       // malloc() allocate the memory for 5 integers     // containing garbage values     arr = (int \*)malloc(5 \* sizeof(int)); // 5\*4bytes = 20 bytes     // Deallocates memory previously allocated by malloc() function     free( arr );     // calloc() allocate the memory for 5 integers and     // set 0 to all of them     arr = (int \*)calloc(5, sizeof(int));     // Deallocates memory previously allocated by calloc() function     free(arr);       return(0);  } |

Copy CodeRun on IDE

We can achieve same functionality as calloc() by using malloc() followed by memset(),

|  |
| --- |
| ptr = malloc(size);  memset(ptr, 0, size); |

Copy Code

***Note:*It would be better to use malloc over calloc, unless we want the zero-initialization because malloc is faster than calloc. So if we just want to copy some stuff or do something that doesn’t require filling of the blocks with zeros, then malloc would be a better choice.**