1. Difference between #define (micro) AND Typedef ?
2. ***#define is a preprocessor token: the compiler itself will never see it.***

***#define is a pre-processor instruction, that creates a text replacement event prior to compilation. When the compiler gets to the code, the original "#defined" word is no longer there. #define is mostly used for macros and global constants.***

***It means before compilation, preprocessor will find all macros***

***And replace them with their original syntax –***

***#define: is a C directive which is used to #define alias.***

|  |
| --- |
| ***// C program to demonstrate #define***  ***#include <stdio.h>***    ***#define HYD "Hyderabad" // After this line HYD is replaced by***  ***// "Hyderabad"***    ***int main()***  ***{***  ***printf("%s ", HYD);***  ***return 0;***  ***}*** |
|  |
|  |

***Output :***

***Hyderabad***

EX.

#define PTR char\*

PTR x, y, z;

***The second line efficiently becomes***

Char\* x, y, z; hopefully you see the problem; only “x” will have the type int\*, y & z will be declared

As Plain “int” (because the \* is associated with the declarator, not the type specifier).

***This makes x and y, z different as x is pointer-to-a char whereas y, z are char variables. When we declare macros with pointers while defining if while defining if declare more than one identifier then actual definition is given to first identifier and for the rest non pointer definition is given. In the above case x will be declared as char\*, so its size is the size of pointer whereas y and z will be declared as char so, there size will be 1 byte.***

***ii> typedef is a compiler token: the preprocessor does not care about it.***

[***Typedef***](http://msdn.microsoft.com/en-us/library/ms221152.aspx)***is a C keyword that creates an alias for a type.***

***Mostly it is provide one level of abstraction for code development.***

***define :when your code is extremely big, scattered across many files,***

***it's better to use #define; it helps in readability***

**typedef**: typedef is used to give data type a new name, for example

|  |
| --- |
| // C program to demonstrate typedef  #include <stdio.h>    typedef unsigned char BYTE; // After this line BYTE can be used  // in place of unsigned char    int main()  {      BYTE b1, b2;      b1 = 'c';      printf("%c ", b1);      return 0;  }  Ex: |

typedef char\* ptr;

ptr a, b, c;

the second line effectively becomes

char\* a, b, c;

***This declares a, b, c all variable as char. pointer***

***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).***

**typedef vs #define**

* **typedef** is limited to giving symbolic names to types only where as **#define** can be used to define alias for values as well, you can define 1 as ONE etc.
* **typedef** interpretation is performed by the **compiler** whereas **#define** statements are processed by the pre-**processor**.

1. Difference between macro and 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.***

***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.*** |

***When the program executes the function call instruction the CPU stores the memory address of the instruction following the function call, copies the arguments of the function on the stack and finally transfers control to the specified function. The CPU then executes the function code, stores the function return value in a predefined memory location/register and returns control to the calling function. This can become overhead if the execution time of function is less than the switching time from the caller function to called function. For functions that are large and/or perform complex tasks, the overhead of the function call is usually insignificant compared to the amount of time the function takes to run. However, for small, commonly-used functions, the time needed to make the function call is often a lot more than the time needed to actually execute the function’s code. This overhead occurs for small functions because execution time of small function is less than the switching time.***

***An inline functions to reduce the function call overhead. Inline function is a function that is expanded in line when it is called. When the inline function is called whole code of the inline function gets inserted or substituted at the point of inline function call. This substitution is performed by the C++ compiler at compile time. Inline function may increase efficiency if it is small.  
The syntax for defining the function inline is:***

***Remember, inlining is only a request to the compiler, not a command. Compiler can ignore the request for inlining. Compiler may not perform inlining in such circumstances***

***like:  
1)If a function contains a loop. (for, while, do-while)***

***2) If a function contains static variables.***

***3) If a function is recursive.***

***4) If a function return type is other than void, and the return statement doesn’t exist in function body.***

***5) If a function contains switch or goto statement.***

***Inline functions provide following advantages:***

***1) Function call overhead doesn’t occur.***

***2) It also saves the overhead of push/pop variables on the stack when function is called.***

***3) It also saves overhead of a return call from a function***

***.  
4) When you inline a function, you may enable compiler to perform context specific optimization on the body of function. Such optimizations are not possible for normal function calls. Other optimizations can be obtained by considering the flows of calling context and the called context.***

***5) Inline function may be useful (if it is small) for embedded systems because inline can yield less code than the function call preamble and return.***

***Inline function disadvantages:***

***1) The added variables from the inlined function consumes additional registers, After in-lining function if variables number which are going to use register increases than they may create overhead on register variable resource utilization. This means that when inline function body is substituted at the point of function call, total number of variables used by the function also gets inserted. So the number of register going to be used for the variables will also get increased. So if after function inlining variable numbers increase drastically then it would surely cause an overhead on register utilization.***

***2) If you use too many inline functions then the size of the binary executable file will be large, because of the duplication of same code.***

***3) Too much inlining can also reduce your instruction cache hit rate, thus reducing the speed of instruction fetch from that of cache memory to that of primary memory.***

***4) Inline function may increase compile time overhead if someone changes the code inside the inline function then all the calling location has to be recompiled because compiler would require to replace all the code once again to reflect the changes, otherwise it will continue with old functionality.***

***5) Inline functions may not be useful for many embedded systems. Because in embedded systems code size is more important than speed.***

***6) Inline functions might cause thrashing because inlining might increase size of the binary executable file. Thrashing in memory causes performance of computer to degrade.***

***The following program demonstrates the use of use of inline function.***

|  |
| --- |
| ***#include <iostream>***  ***using namespace std;***  ***inline int cube(int s)***  ***{***  ***return s\*s\*s;***  ***}***  ***int main()***  ***{***  ***cout << "The cube of 3 is: " << cube(3) << "\n";***  ***return 0;***  ***} //Output: The cube of 3 is: 27*** |

## Macros

***A Macro is typically an abbreviated name given to a piece of code or a value. Macros can also be defined without any value or piece of code but in that case they are used only for testing purpose.***

***Sometimes while programming, we stumble upon a condition where we want to use a value or a small piece of code many times in a code. Also there is a possibility that the in future, the piece of code or value would change. Then changing the value all over the code does not make any sense. There has to be a way out through which one can make the change at one place and it would get reflected at all the places. This is where the concept of a macro fits in.***

***#define is a pre-processor instruction,***

***Lets understand the concept of macros using some example codes.***

***What is wrong with macro?  
Reader’s familiar with the C language knows that C language uses macro. The preprocessor replace all macro calls directly within the macro code. It is recommended to always use inline function instead of macro. According to Dr. Bjarne Stroustrup the creator of C++ that macros are almost never necessary in C++ and they are error prone. There are some problems with the use of macros in C++. Macro cannot access private members of class. Macros looks like function call but they are actually not.***

### **Defining Macros without values**

***The most basic use of macros is to define them without values and use them as testing conditions. As an example, let’s look at the following piece of code:***

#include <stdio.h>

#define MACRO1

#define MACRO2

Int main (void)

{

#ifdef MACRO1 // test whether MACRO1 is defined...

printf("\nMACRO1 Defined\n");

#endif

#ifdef MACRO2 // test whether MACRO2 is defined...

printf("\nMACRO2 Defined\n");

#endif

return 0;

}

* ***So, the above code just defines two macros MACRO1 and MACRO2.***
* ***As clear from the definition, the macros are without any values***
* ***Inside the main function, the macros are used only in testing conditions.***

***Now, if we look at the output, we will see :***

$ ./macro

MACRO1 Defined

MACRO2 Defined

* ***Since both of the macros are defined so both the printf statements executed.***
* ***Now, one would question where these testing macros are used. Well, mostly these type of testing macros are used in a big project involving many source and header files. In such big projects, to avoid including a single header more than once (directly and indirectly through another header file) a macro is defined in the original header and this macro is tested before including the header anywhere so as to be sure that if the macros is already defined then there is no need to include the header as it has already been included (directly or indirectly).***

***Dynamic memory allocation***

# Difference Between malloc and calloc

***Dynamic Memory Allocation (Malloc, calloc, realloc, free)***

| **Function** | **Use of Function** |
| --- | --- |
| [**malloc()**](http://www.programiz.com/c-programming/c-dynamic-memory-allocation#malloc) | * **memory allocation --ptr=(int\*)malloc(100\*sizeof(int));// single argument** * **Allocates single block of memory** * **Return a pointer of type void, so type casting is done** * Does not initialize the memory allocated. * Takes one argument that is, **number of bytes/ (size of ().)** * malloc is faster than calloc |
| [**calloc()**](http://www.programiz.com/c-programming/c-dynamic-memory-allocation#calloc) | * **contiguous allocation** -- **ptr=(int\*)calloc(n,element-size);//two argument** * **Calloc() allocates multiple blocks of memory each of same size** * **Returns a pointer to memory** * **Initializes the allocated memory to ZERO.** * **Take two arguments :** number of blocks and size of each block (Depends on data type) * calloc takes little longer than malloc because of the extra step of initializing the allocated memory by zero |
| [**free()**](http://www.programiz.com/c-programming/c-dynamic-memory-allocation#free) | dellocate the previously allocated space |
| [**realloc()**](http://www.programiz.com/c-programming/c-dynamic-memory-allocation#realloc) | **Reallocation--ptr=realloc(ptr,newsize)**  Change the size of previously allocated space  If the previously allocated memory is insufficient or more than sufficient. Then, you can change memory size previously allocated using realloc(). |

|  |  |
| --- | --- |
| Differences between malloc and calloc | |
| **malloc** | **calloc** |
| The name malloc stands for *memory allocation*. | The name calloc stands for *contiguous allocation*. |
| void \*malloc(size\_t n) returns a pointer to n bytes of uninitialized storage, or NULL if the request cannot be satisfied. If the space assigned by malloc() is overrun, the results are undefined. | void \*calloc(size\_t n, size\_t size)returns a pointer to enough free space for an array of n objects of the specified size, or NULL if the request cannot be satisfied. The storage is initialized to zero. |
| malloc() takes one argument that is, *number of bytes*. | calloc() take two arguments those are: *number of blocks* and *size of each block*. |
| syntax of malloc():  void \*malloc(size\_t n);  Allocates n bytes of memory. If the allocation succeeds, a void pointer to the allocated memory is returned. Otherwise NULL is returned. | syntax of calloc():  void \*calloc(size\_t n, size\_t size);  Allocates a contiguous block of memory large enough to hold n elements of sizebytes each. The allocated region is initialized to zero. |
| malloc is faster than calloc. | calloc takes little longer than mallocbecause of the extra step of initializing the allocated memory by zero. However, in practice the difference in speed is very tiny and not recognizable. |

## Similarities Between Malloc and Calloc

***The pointer returned by malloc*** or calloc ***has the proper alignment for the object in question, but it must be cast into the appropriate type.***

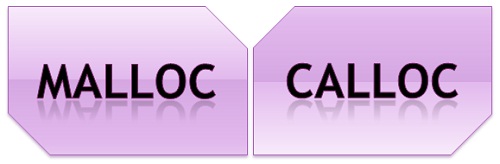
***Proper alignment means the value of the returned address is guaranteed to be an even multiple of alignment. The value of alignment must be a power of two and must be greater than or equal to the size of a word.***

The malloc(), calloc() f***unctions will fail if:***

* ***The physical limits of the system are exceeded by n bytes of memory which cannot be allocated.***
* ***There is not enough memory available to allocate n bytes of memory; but the application could try again later.***

# Difference Between malloc and calloc

July 20, 2017 [Leave a Comment](http://techdifferences.com/difference-between-malloc-and-calloc.html#respond)

The fundamental difference between malloc and calloc function is that **calloc()**needs two arguments instead of one argument which is required by **malloc()**. Both malloc() and calloc() are the functions which C programming language provides for dynamic memory allocation and de-allocation at run time.

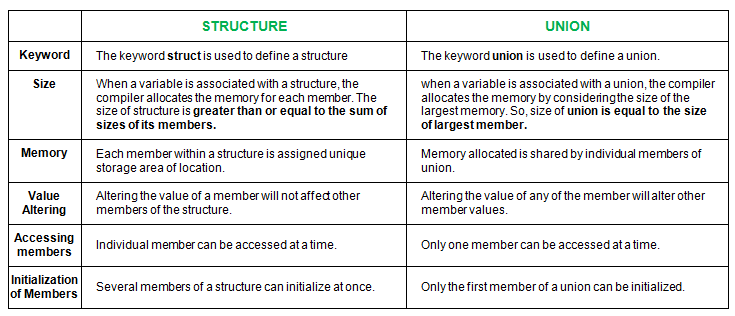
Before understanding malloc() and calloc() functions first let us understand meaning of dynamic memory allocation. **Memory allocation** is the procedure of assigning the computer memory for the execution of programs and processes. We use dynamic allocation techniques, when it is not known in advance how much of memory space is needed for the program and process.

Dynamic memory allocation arises due to the problems associated with static memory allocation such as if fewer elements are stored, then the rest of the memory is unnecessarily wasted. Therefore, it overcomes the problems of static memory allocation where memory is allocated only when it is required.

| **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

**Differences**



|  |  |
| --- | --- |
| 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. |
| **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;  }; | **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.

|  |
| --- |
| 1. // C program to illustrate differences 2. // between structure and Union 3. #include <stdio.h> 4. #include <string.h> 6. // declaring structure 7. struct struct\_example 8. { 9. int integer; 10. float decimal; 11. char name[20]; 12. }; 14. // declaraing union 16. union union\_example 17. { 18. int integer; 19. float decimal; 20. char name[20]; 21. }; 23. void main() 24. { 25. // creating variable for structure 26. // and initializing values difference 27. // six 28. struct struct\_example s={18,38,"geeksforgeeks"}; 30. // creating variable for union 31. // and initializing values 32. union union\_example u={18,38,"geeksforgeeks"};  35. printf("structure data:\n integer: %d\n" 36. "decimal: %.2f\n name: %s\n", 37. s.integer, s.decimal, s.name); 38. printf("\nunion data:\n integeer: %d\n" 39. "decimal: %.2f\n name: %s\n", 40. u.integer, u.decimal, u.name);  43. // difference two and three 44. printf("\nsizeof structure : %d\n", sizeof(s)); 45. printf("sizeof union : %d\n", sizeof(u)); 47. // difference five 48. printf("\n Accessing all members at a time:"); 49. s.integer = 183; 50. s.decimal = 90; 51. strcpy(s.name, "geeksforgeeks"); 53. printf("structure data:\n integer: %d\n " 54. "decimal: %.2f\n name: %s\n", 55. s.integer, s.decimal, s.name); 57. u.integer = 183; 58. u.decimal = 90; 59. strcpy(u.name, "geeksforgeeks"); 61. printf("\nunion data:\n integeer: %d\n " 62. "decimal: %.2f\n name: %s\n", 63. u.integer, u.decimal, u.name); 65. printf("\n Accessing one member at time:"); 67. printf("\nstructure data:"); 68. s.integer = 240; 69. printf("\ninteger: %d", s.integer); 71. s.decimal = 120; 72. printf("\ndecimal: %f", s.decimal); 74. strcpy(s.name, "C programming"); 75. printf("\nname: %s\n", s.name); 77. printf("\n union data:"); 78. u.integer = 240; 79. printf("\ninteger: %d", u.integer); 81. u.decimal = 120; 82. printf("\ndecimal: %f", u.decimal); 84. strcpy(u.name, "C programming"); 85. printf("\nname: %s\n", u.name); 87. //difference four 88. printf("\nAltering a member value:\n"); 89. s.integer = 1218; 90. printf("structure data:\n integer: %d\n " 91. " decimal: %.2f\n name: %s\n", 92. s.integer, s.decimal, s.name); 94. u.integer = 1218; 95. printf("union data:\n integer: %d\n" 96. " decimal: %.2f\n name: %s\n", 97. u.integer, u.decimal, u.name); 98. } |

Run 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: ?

In my opinion, structure is better because as memory is shared in union ambiguity is more.

# Structures in C

# 1>A structure is a user defined data type in C.

# A structure creates a data type that can be used to group items of possibly

# Different DATA types into a single type.

***How to create a structure?***

***‘struct’ keyword is used to create a structure. Following is an example.***

|  |
| --- |
| struct addrress  {     char name[50];     char street[100];     char city[50];     char state[20]     int pin;  }; |

***How to declare structure variables?***

***A structure variable can either be declared with structure declaration or as a separate declaration like basic types.***

|  |
| --- |
| // A variable declaration with structure declaration.  struct Point  {     int x, y;  } p1;  // The variable p1 is declared with 'Point'      // A variable declaration like basic data types  struct Point  {     int x, y;  };    int main()  {     struct Point p1;  // The variable p1 is declared like a normal variable  } |

***Note: In C++, the struct keyword is optional before in declaration of variable. In C, it is mandatory.***

***How to initialize structure members?***

***Structure members cannot be initialized with declaration. For example the following C program fails in compilation.***

|  |
| --- |
| struct Point  {     int x = 0;  // COMPILER ERROR:  cannot initialize members here     int y = 0;  // COMPILER ERROR:  cannot initialize members here  }; |

Run on IDE

***The reason for above error is simple, when a datatype is declared, no memory is allocated for it. Memory is allocated only when variables are created.***

***Structure members can be initialized using curly braces ‘{}’. For example, following is a valid initialization.***

|  |
| --- |
| struct Point  {     int x, y;  };    int main()  {     // A valid initialization. member x gets value 0 and y     // gets value 1.  The order of declaration is followed.     struct Point p1 = {0, 1};  } |

***How to access structure elements?***

|  |  |
| --- | --- |
| Structure members are accessed using dot (.) operator. | ***What is a structure pointer?*** Like primitive types, we can have pointer to a structure. If we have a pointer to structure, members are accessed using arrow ( -> ) operator. |
| struct Point  {     int x, y;  };    int main()  {     struct Point p1 = {0, 1};       // Accesing members of point p1     p1.x = 20;     printf ("x = %d, y = %d", p1.x, p1.y); | struct Point  {     int x, y;  };    int main()  {     struct Point p1 = {1, 2};       // p2 is a pointer to structure p1     struct Point \*p2 = &p1;       // Accessing structure members using structure pointer     printf("%d %d", p2->x, p2->y);     return 0;  } |
|  | struct Point  {     int x, y;  }\*p,x; // p is a pointer to structure x is object to structure    int main()  {  p->x=10;  p->y=20;     // Accessing structure members using structure pointer     printf("%d %d", p->x, p->y);     return 0;  } |

**What is designated Initialization?**

***Designated Initialization allows structure members to be initialized in any order. This feature has been added in***[***C99 standard***](http://www.geeksforgeeks.org/c-programming-language-standard/)***.***

|  |
| --- |
| ***struct Point***  ***{***  ***int x, y, z;***  ***};***    ***int main()***  ***{***  ***// Examples of initializtion using designated initialization***  ***struct Point p1 = {.y = 0, .z = 1, .x = 2};***  ***struct Point p2 = {.x = 20};***    ***printf ("x = %d, y = %d, z = %d\n", p1.x, p1.y, p1.z);***  ***printf ("x = %d", p2.x);***  ***return 0;***  ***}*** |

Run on IDE

Output:

x = 2, y = 0, z = 1

x = 20

***This feature is not available in C++ and works only in C.***

**What is an array of structures?**

***Like other primitive data types, we can create an array of structures***.

|  |
| --- |
| struct Point  {     int x, y;  };    int main()  {     // Create an array of structures     struct Point arr[10];       // Access array members     arr[0].x = 10;     arr[0].y = 20;       printf("%d %d", arr[0].x, arr[0].y);     return 0;  } |

Run on IDE

Output:

10 20

# [Difference between typedef “struct” and “struct”?](https://stackoverflow.com/questions/36057712/difference-between-typedef-struct-and-struct)

1. ***I know the basic difference between them just have a doubt in particular situation like following:***
2. struct books{
3. int id;
4. char\* title;
5. }book;
6. book.id=9; // this is valid;
7. ***But in case of typedef :***
8. typedef struct books{
9. int id;
10. char\*title;
11. }book;
12. book.id=9; //it is not valid we have to do like book b1; then b1.id=9 is valid

***In the second, you are defining an alias book for the type struct books. Thus book is not an object but a type name***

# Union in C

* ***Like***[***Structures***](http://quiz.geeksforgeeks.org/structures-c/)***, union is a user defined data type. In union, all members share the same memory location. For example in the following C program, both x and y share the same location. If we change x, we can see the changes being reflected in y.***

|  |
| --- |
| #include <stdio.h>    // Declaration of union is same as structures  union test  {     int x, y;  };    int main()  {       // A union variable t      union test t;        t.x = 2; // t.y also gets value 2      printf ("After making x = 2:\n x = %d, y = %d\n\n",               t.x, t.y);        t.y = 10;  // t.x is also updated to 10      printf ("After making Y = 'A':\n x = %d, y = %d\n\n",               t.x, t.y);      return 0;  } |

* Output:
* After making x = 2:
* x = 2, y = 2
* After making Y = 'A':
* x = 10, y = 10

***How is the size of union decided by compiler?***

***Size of a union is taken according the size of largest member in union.***

|  |
| --- |
| #include <stdio.h>    union test1  {     int x;     int y;  };    union test2  {     int x;     char y;  };    union test3  {     int arr[10];     char y;  };    int main()  {      printf ("sizeof(test1) = %d, sizeof(test2) = %d,"              "sizeof(test3) =  %d", sizeof(test1),              sizeof(test2), sizeof(test3));      return 0;  } |

* Run on IDE
* Output
* sizeof(test1) = 4, sizeof(test2) = 4,sizeof(test3) = 40

**Pointers to unions?**

***Like structures, we can have pointers to unions and can access members using arrow operator (->). The following example demonstrates the same.***

|  |
| --- |
| union test  {     int x;     char y;  };    int main()  {     union test p1;     p1.x = 65;       // p2 is a pointer to union p1     union test \*p2 = &p1;       // Accessing union members using pointer     printf("%d %c", p2->x, p2->y);     return 0;  } |

* Run on IDE
* 65 A

**What are applications of union?**

***Unions can be useful in many situations where we want to use same memory for two ore more members. For example, suppose we want to implement a binary tree data structure where each leaf node has a double data value, while each internal node has pointers to two children, but no data. If we declare this as:***

|  |
| --- |
| struct NODE {    struct NODE \*left;    struct NODE \*right;    double data;  }; |

* ***then every node requires 16 bytes, with half the bytes wasted for each type of node. On the other hand, if we declare a node as following, then we can save space.***

|  |
| --- |
| struct NODE  {      bool is\_leaf;      union      {          struct          {              struct NODE \*left;              struct NODE \*right;          } internal;          double data;      } info;  }; |

# 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*** |

**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*** |

***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*** |

***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*** |

***6) A token passed to macro can be converted to a sting 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*** |

***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*** |

***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;***  ***}*** |

***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;***  ***}*** |

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

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

***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 \*/*** |

# 5> Segmentation Fault (SIGSEGV) vs Bus Error (SIGBUS)

**Segmentation fault (SIGSEGV)** and **Bus error(SIGBUS)** are signals generated when serious program error is detected by the operating system and there is no way the program could continue to execute because of these errors.

**1)**[**Segmentation Fault**](http://www.geeksforgeeks.org/core-dump-segmentation-fault-c-cpp/) (also known as SIGSEGV and is usually signal 11) occur when the program tries to write/read outside the memory allocated for it or when writing memory which can only be read.In other words when the program tries to access the memory to which it doesn’t have access to. SIGSEGV is

Abbreviation for “Segmentation Violation”

. 

Few cases where SIGSEGV signal generated are as follows,

-> Using uninitialized pointer

-> De-referencing a NULL pointer

-> Trying to access memory that the program doesn’t own (eg. trying to access an array element

out of array bounds).

-> Trying to access memory which is already de-allocated (trying to use dangling pointers).

Please refer [this](http://www.geeksforgeeks.org/core-dump-segmentation-fault-c-cpp/) article for examples

.

**2)**[**Bus Error**](https://en.wikipedia.org/wiki/Bus_error) (also known as SIGBUS and is usually signal 10) occur when a process is trying to access memory that the CPU cannot physically address.In other words the memory tried to access by the program is not a valid memory address.It caused due to alignment issues with the CPU (eg. trying to read a long from an address which isn’t a multiple of 4). SIGBUS is abbrivation for “Bus Error”.

SIGBUS signal occurs in below cases

,-> Program instructs the CPU to read or write a specific physical memory address which is not valid

/ Requested physical address is unrecognized by the whole computer system

.  
-> Unaligned access of memory (For example, if multi-byte accesses must be 16 bit-aligned, addresses

(given in bytes) at 0, 2, 4, 6, and so on would be considered aligned and therefore accessible, while

addresses 1, 3, 5, and so on would be considered unaligned.)

**The main difference** between Segmentation Fault and Bus Error is that Segmentation Fault indicates an invalid access to a valid memory, while Bus Error indicates an access to an invalid address.

# 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 |

## **Call by value in C**

In call by value, **original value is not modified**.

In call by value, value being passed to the function is locally stored by the function parameter in stack memory location. If you change the value of function parameter, it is changed for the current function only. It will not change the value of variable inside the caller method such as main().

Let's try to understand the concept of call by value in c language by the example given below:

1. #include <stdio.h>
2. #include <conio.h>
3. **void** change(**int** num) {
4. printf("Before adding value inside function num=%d \n",num);
5. num=num+100;
6. printf("After adding value inside function num=%d \n", num);
7. }
9. **int** main() {
10. **int** x=100;
11. clrscr();
13. printf("Before function call x=%d \n", x);
14. change(x);//passing value in function
15. printf("After function call x=%d \n", x);
17. getch();
18. **return** 0;
19. }

#### **Output**

Before function call x=100

Before adding value inside function num=100

After adding value inside function num=200

## **Call by reference in C**

In call by reference, **original value is modified** because we pass reference (address).

Here, address of the value is passed in the function, so actual and formal arguments shares the same address space. Hence, value changed inside the function, is reflected inside as well as outside the function.

Note: To understand the call by reference, you must have the basic knowledge of pointers.

Let's try to understand the concept of call by reference in c language by the example given below:

1. #include <stdio.h>
2. #include <conio.h>
3. **void** change(**int** \*num) {
4. printf("Before adding value inside function num=%d \n",\*num);
5. (\*num) += 100;
6. printf("After adding value inside function num=%d \n", \*num);
7. }
9. **int** main() {
10. **int** x=100;
11. clrscr();
13. printf("Before function call x=%d \n", x);
14. change(&x);//passing reference in function
15. printf("After function call x=%d \n", x);
17. getch();
18. **return** 0;
19. }

#### **Output**

Before function call x=100

Before adding value inside function num=100

After adding value inside function num=200

After function call x=200

## 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.

* 1. What is the purpose of extern storage specifier?

Used to resolve the scope of global symbol.

Eg:

main() {

extern int i; //here you only declare it, “I” is a global variable define in

Other .c file if you write extern int i=10 it give error but you can do operation on it like i=20,x=I,

Printf(“%d”,i);

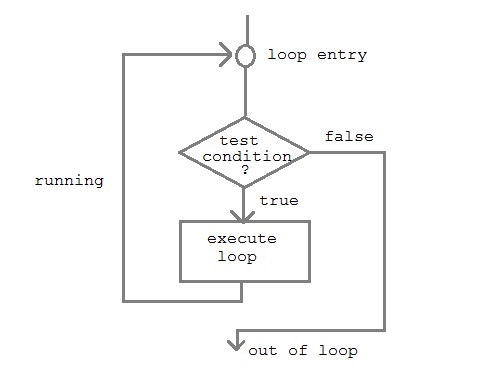
}

int i = 20;

# loop in C programming with example

A loop is used for executing a block of statements repeatedly until a given condition returns false.

#### How it Works



A sequence of statements are executed until a specified condition is true. This sequence of statements to be executed is kept inside the curly braces { } known as the **Loop body**. After every execution of loop body, condition is verified, and if it is found to be **true** the loop body is executed again. When the condition check returns **false**, the loop body is not executed.

#### There are 3 type of Loops in C language

1. *while* loop
2. *for* loop
3. *do-while* loop

You may encounter situations, when a block of code needs to be executed several number of times. In general, statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on.

Programming languages provide various control structures that allow for more complicated execution paths.

A loop statement allows us to execute a statement or group of statements multiple times. Given below is the general form of a loop statement in most of the programming languages −



C programming language provides the following types of loops to handle looping requirements.

|  |  |
| --- | --- |
| **S.N.** | **Loop Type & Description** |
| 1 | [**while loop**](https://www.tutorialspoint.com/cprogramming/c_while_loop.htm)  Repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body. |
| 2 | [**for loop**](https://www.tutorialspoint.com/cprogramming/c_for_loop.htm)  Executes a sequence of statements multiple times and abbreviates the code that manages the loop variable. |
| 3 | [**do...while loop**](https://www.tutorialspoint.com/cprogramming/c_do_while_loop.htm)  It is more like a while statement, except that it tests the condition at the end of the loop body. |
| 4 | [**nested loops**](https://www.tutorialspoint.com/cprogramming/c_nested_loops.htm)  You can use one or more loops inside any other while, for, or do..while loop. |

## **Loop Control Statements**

Loop control statements change execution from its normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed.

C supports the following control statements.

|  |  |
| --- | --- |
| **S.N.** | **Control Statement & Description** |
| 1 | [**break statement**](https://www.tutorialspoint.com/cprogramming/c_break_statement.htm)  Terminates the **loop** or **switch** statement and transfers execution to the statement immediately following the loop or switch. |
| 2 | [**continue statement**](https://www.tutorialspoint.com/cprogramming/c_continue_statement.htm)  Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating. |
| 3 | [**goto statement**](https://www.tutorialspoint.com/cprogramming/c_goto_statement.htm)  Transfers control to the labeled statement. |

## **The Infinite Loop**

A loop becomes an infinite loop if a condition never becomes false. The forloop is traditionally used for this purpose. Since none of the three expressions that form the 'for' loop are required, you can make an endless loop by leaving the conditional expression empty.

#include <stdio.h>

int main () {

for( ; ; ) {

printf("This loop will run forever.\n");

}

return 0;

}

When the conditional expression is absent, it is assumed to be true. You may have an initialization and increment expression, but C programmers more commonly use the for(;;) construct to signify an infinite loop.

FOR Loop:

## **Syntax**

The syntax of a **for** loop in C programming language is −

for ( init; condition; increment ) {

statement(s);

}

Here is the flow of control in a 'for' loop −

* The **init** step is executed first, and only once. This step allows you to declare and initialize any loop control variables. You are not required to put a statement here, as long as a semicolon appears.
* Next, the **condition** is evaluated. If it is true, the body of the loop is executed. If it is false, the body of the loop does not execute and the flow of control jumps to the next statement just after the 'for' loop.
* After the body of the 'for' loop executes, the flow of control jumps back up to the **increment** statement. This statement allows you to update any loop control variables. This statement can be left blank, as long as a semicolon appears after the condition.
* The condition is now evaluated again. If it is true, the loop executes and the process repeats itself (body of loop, then increment step, and then again condition). After the condition becomes false, the 'for' loop terminates.

## **Flow Diagram**



## **Example**

#include <stdio.h>

int main () {

int a;

/\* for loop execution \*/

for( a = 10; a < 20; a = a + 1 ){

printf("value of a: %d\n", a);

}

return 0;

}

# while loop in C

A **while** loop in C programming repeatedly executes a target statement as long as a given condition is true.

## **Syntax**

The syntax of a **while** loop in C programming language is −

while(condition) {

statement(s);

}

Here, **statement(s)** may be a single statement or a block of statements. The **condition** may be any expression, and true is any nonzero value. The loop iterates while the condition is true.

When the condition becomes false, the program control passes to the line immediately following the loop.

## **Flow Diagram**



Here, the key point to note is that a while loop might not execute at all. When the condition is tested and the result is false, the loop body will be skipped and the first statement after the while loop will be executed.

## **Example**

#include <stdio.h>

int main () {

/\* local variable definition \*/

int a = 10;

/\* while loop execution \*/

while( a < 20 ) {

printf("value of a: %d\n", a);

a++;

}

return 0;

}

# do...while loop in C

Unlike **for** and **while** loops, which test the loop condition at the top of the loop, the **do...while** loop in C programming checks its condition at the bottom of the loop.

A **do...while** loop is similar to a while loop, except the fact that it is guaranteed to execute at least one time.

## **Syntax**

The syntax of a **do...while** loop in C programming language is −

do {

statement(s);

} while( condition );

Notice that the conditional expression appears at the end of the loop, so the statement(s) in the loop executes once before the condition is tested.

If the condition is true, the flow of control jumps back up to do, and the statement(s) in the loop executes again. This process repeats until the given condition becomes false.

## **Flow Diagram**



## **Example**

#include <stdio.h>

int main () {

/\* local variable definition \*/

int a = 10;

/\* do loop execution \*/

do {

printf("value of a: %d\n", a);

a = a + 1;

}while( a < 20 );

return 0;

}

# C - if statement

An **if** statement consists of a Boolean expression followed by one or more statements.

## **Syntax**

The syntax of an 'if' statement in C programming language is −

if(boolean\_expression) {

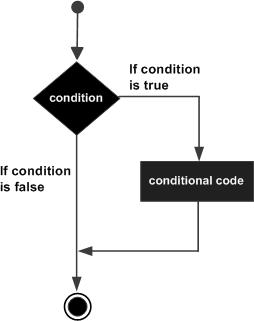
/\* statement(s) will execute if the boolean expression is true \*/

}

If the Boolean expression evaluates to **true**, then the block of code inside the 'if' statement will be executed. If the Boolean expression evaluates to **false**, then the first set of code after the end of the 'if' statement (after the closing curly brace) will be executed.

C programming language assumes any **non-zero** and **non-null** values as **true** and if it is either **zero** or **null**, then it is assumed as **false** value.

## **Flow Diagram**



## **Example**

#include <stdio.h>

int main () {

/\* local variable definition \*/

int a = 10;

/\* check the boolean condition using if statement \*/

if( a < 20 ) {

/\* if condition is true then print the following \*/

printf("a is less than 20\n" );

}

printf("value of a is : %d\n", a);

return 0;

}

When the above code is compiled and executed, it produces the following result −

a is less than 20;

value of a is : 10

# C - if...else statement

Advertisements

[Previous Page](https://www.tutorialspoint.com/cprogramming/c_decision_making.htm)

[Next Page](https://www.tutorialspoint.com/cprogramming/c_decision_making.htm)

An **if** statement can be followed by an optional **else** statement, which executes when the Boolean expression is false.

## **Syntax**

The syntax of an **if...else** statement in C programming language is −

if(boolean\_expression) {

/\* statement(s) will execute if the boolean expression is true \*/

}

else {

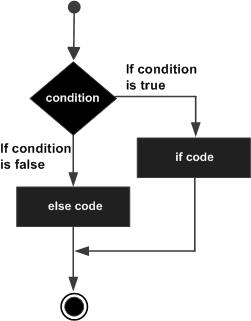
/\* statement(s) will execute if the boolean expression is false \*/

}

If the Boolean expression evaluates to **true**, then the **if block** will be executed, otherwise, the **else block** will be executed.

C programming language assumes any **non-zero** and **non-null** values as **true**, and if it is either **zero** or **null**, then it is assumed as **false** value.

## **Flow Diagram**



## **Example**

#include <stdio.h>

int main () {

/\* local variable definition \*/

int a = 100;

/\* check the boolean condition \*/

if( a < 20 ) {

/\* if condition is true then print the following \*/

printf("a is less than 20\n" );

}

else {

/\* if condition is false then print the following \*/

printf("a is not less than 20\n" );

}

printf("value of a is : %d\n", a);

return 0;

}

When the above code is compiled and executed, it produces the following result −

a is not less than 20;

value of a is : 100

## **If...else if...else Statement**

An **if** statement can be followed by an optional **else if...else** statement, which is very useful to test various conditions using single if...else if statement.

When using if...else if..else statements, there are few points to keep in mind −

* An if can have zero or one else's and it must come after any else if's.
* An if can have zero to many else if's and they must come before the else.
* Once an else if succeeds, none of the remaining else if's or else's will be tested.

### **Syntax**

The syntax of an **if...else if...else** statement in C programming language is −

if(boolean\_expression 1) {

/\* Executes when the boolean expression 1 is true \*/

}

else if( boolean\_expression 2) {

/\* Executes when the boolean expression 2 is true \*/

}

else if( boolean\_expression 3) {

/\* Executes when the boolean expression 3 is true \*/

}

else {

/\* executes when the none of the above condition is true \*/

}

### **Example**

#include <stdio.h>

int main () {

/\* local variable definition \*/

int a = 100;

/\* check the boolean condition \*/

if( a == 10 ) {

/\* if condition is true then print the following \*/

printf("Value of a is 10\n" );

}

else if( a == 20 ) {

/\* if else if condition is true \*/

printf("Value of a is 20\n" );

}

else if( a == 30 ) {

/\* if else if condition is true \*/

printf("Value of a is 30\n" );

}

else {

/\* if none of the conditions is true \*/

printf("None of the values is matching\n" );

}

printf("Exact value of a is: %d\n", a );

return 0;

}

When the above code is compiled and executed, it produces the following result −

None of the values is matching

Exact value of a is: 100

* 1. TYPE CASTING IN C LANGUAGE

***=============***Type casting is a way to convert a variable from one data type to another data type. For example, if you want to store a long value into a simple integer then you can type cast long to int. You can convert values from one type to another explicitly using the cast operator.

New data type should be mentioned before the variable name or value in brackets which to be typecast.

# WHAT IS TYPE CASTING IN C LANGUAGE?

* Converting an expression of a given type into another type is known as type-casting . typecasting is more use in c language programming.
* Here, It is best practice to convert lower data type to higher data type to avoid data loss.
* Data will be truncated when higher data type is converted to lower. For example, if float is converted to int, data which is present after decimal point will be lost.
* **There are two types of type casting in c language.**

## TYPES OF TYPECASTING IN C

|  |  |
| --- | --- |
| S.No | Types of typecasting in C Programming |
| 1 | Implicit Conversion |
| 2 | Explicit Conversion |

### **IMPLICIT CONVERSION**

**Implicit Type Conversion** Also known as ‘automatic type conversion’.

* Done by the compiler on its own, without any external trigger from the user.
* Generally takes place when in an expression more than one data type is present. In such condition type conversion (type promotion) takes place to avoid lose of data.
* All the data types of the variables are upgraded to the data type of the variable with largest data type.
* **bool -> char -> short int -> int ->**
* **unsigned int -> long -> unsigned ->**
* **long long -> float -> double -> long double**
* It is possible for implicit conversions to lose information, signs can be lost (when signed is implicitly converted to unsigned), and overflow can occur (when long long is implicitly converted to float).

**Example of Type Implicit Conversion:**

|  |
| --- |
| // An example of implicit conversion  #include<stdio.h>  int main()  {      int x = 10;    // integer x      char y = 'a';  // character c        // y implicitly converted to int. ASCII      // value of 'a' is 97      x = x + y;        // x is implicitly converted to float      float z = x + 1.0;        printf("x = %d, z = %f", x, z);      return 0;  } |

Run on IDE

Output:

x = 107, z = 108.000000

eg 2.

* Implicit conversions do not required any operator for converted . They are automatically performed when a value is copied to a compatible type in program .
* Here, the value of a has been promoted from int to double and we have not had to specify any type-casting operator. This is known as a standard conversion.

**Example :-**

|  |  |
| --- | --- |
| 1 | #include<stdio.h> |
| 2 | #include<conio.h> |
| 3 | void main() |
| 4 | { |
| 5 | int i=20; |
| 6 | double p; |
| 7 | clrscr(); |
| 8 |  |
| 9 | p=i; // implicit conversion |
| 10 |  |
| 11 | printf(“implicit value is %d”,p); |
| 12 |  |
| 13 | getch(); |
| 14 | } |

**Output :-**  
implicit value is 20.

### **EXPLICIT CONVERSION**

* **Explicit Type Conversion**– This process is also called type casting and it is user defined. Here the user can type cast the result to make it of a particular data type.
* The syntax in C:

(type) expression

Type indicated the data type to which the final result is converted.

|  |
| --- |
| // C program to demonstrate explicit type casting  #include<stdio.h>    int main()  {      double x = 1.2;        // Explicit conversion from double to int      int sum = (int)x + 1;        printf("sum = %d", sum);        return 0;  } |

Run on IDE

Output:

sum = 2

Advantages of Type Conversion

* This is done to take advantage of certain features of type hierarchies or type representations.
* It helps us to compute expressions containing variables of different data types.

**Eg2.**

In c language , Many conversions, specially those that imply a different interpretation of the value, require an explicit conversion. We have already seen two notations for explicit type conversion.

They are not automatically performed when a value is copied to a compatible type in program.

**Example :-**

|  |  |
| --- | --- |
| 1 | #include<stdio.h> |
| 2 | #include<conio.h> |
| 3 | void main() |
| 4 | { |
| 5 | int i=20; |
| 6 | short p; |
| 7 | clrscr(); |
| 8 |  |
| 9 | p = (short) i; // Explicit conversion |
| 10 |  |
| 11 | printf(“Explicit value is %d”,p); |
| 12 |  |
| 13 | getch(); |
| 14 | } |

**Output :-**  
Explicit value is 20.

## **Integer Promotion**

Integer promotion is the process by which values of integer type "smaller" than **int** or **unsigned int** are converted either to **int** or **unsigned int**. Consider an example of adding a character with an integer −

#include <stdio.h>

main() {

int i = 17;

char c = 'c'; /\* ascii value is 99 \*/

int sum;

sum = i + c;

printf("Value of sum : %d\n", sum );

}

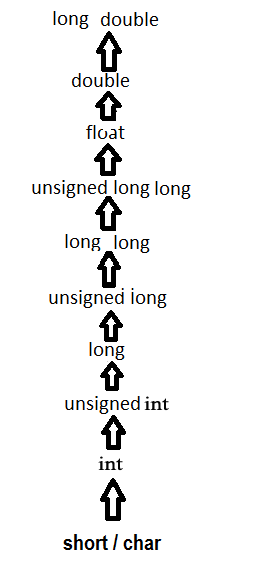
When the above code is compiled and executed, it produces the following

result −

Value of sum : 116

Here, the value of sum is 116 because the compiler is doing integer promotion and converting the value of 'c' to ASCII before performing the actual addition operation.

* **Usual Arithmetic Conversion**
* The usual arithmetic conversions are implicitly performed to cast their values in a common type,C uses the rule that, in all expressions except assignments, any implicit type conversions made from a lower size type to a higher size type as shown below:



* **Inbuilt Typecast Functions In C:**
* There are many inbuilt typecasting functions available in C language which performs data type conversion from one type to another.

|  |  |  |
| --- | --- | --- |
| S.No | TypeCast Function | Description |
| 1 | atof() | Convert string to Float |
| 2 | atoi() | Convert string to int |
| 3 | atol() | Convert string to long |
| 4 | itoa() | Convert int to string |
| 5 | ltoa() | Convert long to string |

* 1. Enumeration (or enum) in C
* **Meaning :e·nu·mer·a·tion**
* **NOUN**

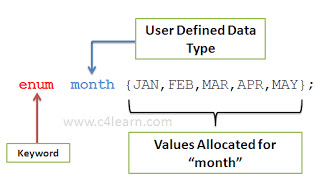
1 an act of [enumerating](http://int.search.myway.com/search/GGmain.jhtml?n=783a814a&p2=%5EAYW%5Exdm591%5ES23657%5Ein&pg=GGmain&pn=1&ptb=D860D340-5480-4EFE-9A3D-4361D34775CD&qs=&searchfor=enumeration+meaning&si=45710064398&ss=sub&st=tab&trs=wtt).

* 1. **a** catalog or list.
* [सूची](http://www.shabdkosh.com/translate/%E0%A4%B8%E0%A5%82%E0%A4%9A%E0%A5%80/%E0%A4%B8%E0%A5%82%E0%A4%9A%E0%A5%80-meaning-in-Hindi-English)
* Enumeration (or enum) is a user defined data type in C.
* It is mainly used to assign names to integral constants
* the names make a program easy to read and maintain.
* enum State {Working = 1, Failed = 0};
* The keyword ‘enum’ is used to declare new enumeration types in C and C++

**Syntax:**

enum identifier {value1, value2,.... Value n};

* enum is **” Enumerated Data Type “**.
* enum is **user defined** data type
* In the above example **“identifier”** is nothing but the **user defined data type** .
* Value1,Value2,Value3….. etc creates **one set of enum values**.
* Using **“identifier”** we are **creating our variables**.



Let’s Take Look at Following Example

Example :

**enum** month {JAN,FEB,MAR,APR,MAY,JUN,JUL,AUG,SEP,OCT,DEC};

**enum** month rmonth;

1. First Line Creates **“User Defined Data Type”** called **month**.
2. It has 12 values as given in the **pair of braces**.
3. In the second line “rmonth” variable is declared of type “month” which can be initialized with any “data value amongst 12 values”.

rmonth = FEB;

1. **Default Numeric value** assigned to first enum value is “0”.
2. Numerically JAN is given value “0”.
3. FEB is given value “1”.
4. MAR is given value “2”.
5. APR is given value “3”.
6. MAY is given value “4”.
7. JUN is given value “5”.
8. and so on…..

printf("%d",rmonth);

* It will Print “1” on the screen because “Numerical Equivalent” of  “FEB” is 1 and “rmonth” is initialized by “FEB”.
* Generally Printing Value of enum variable is as good as printing “**Integer**“.

## **Sample Program :**

#include< stdio.h>

**void** main()

{

**int** i;

**enum** month {JAN,FEB,MAR,APR,MAY,JUN,JUL,AUG,SEP,OCT,DEC};

clrscr();

**for**(i=JAN;i<=DEC;i++)

printf("\n%d",i);

}

**Output :**

01234567891011

Consider –

for(i=JAN;i<=DEC;i++)

Above Statement is

Similar to – [ Replace JAN , DEC with equivalent Numeric Code ]

for(i=0;i<=11;i++)

**Interesting facts about initialization of enum.**

**1.** Two enum names can have same value. For example, in the following C program both ‘Failed’ and ‘Freezed’ have same value 0.

|  |
| --- |
| #include <stdio.h>  enum State {Working = 1, Failed = 0, Freezed = 0};    int main()  {     printf("%d, %d, %d", Working, Failed, Freezed);     return 0;  } |

Run on IDE

Output:

1, 0, 0

**2.** If we do not explicitly assign values to enum names, the compiler by default assigns values starting from 0. For example, in the following C program, sunday gets value 0, monday gets 1, and so on.

|  |
| --- |
| #include <stdio.h>  enum day {sunday, monday, tuesday, wednesday, thursday, friday, saturday};    int main()  {      enum day d = thursday;      printf("The day number stored in d is %d", d);      return 0;  } |

Run on IDE

Output:

The day number stored in d is 4

**3.** We can assign values to some name in any order. All unassigned names get value as value of previous name plus one.

|  |
| --- |
| #include <stdio.h>  enum day {sunday = 1, monday, tuesday = 5,            wednesday, thursday = 10, friday, saturday};    int main()  {      printf("%d %d %d %d %d %d %d", sunday, monday, tuesday,              wednesday, thursday, friday, saturday);      return 0;  } |

Run on IDE

Output:

1 2 5 6 10 11 12

**4.** The value assigned to enum names must be some integeral constant, i.e., the value must be in range from minimum possible integer value to maximum possible integer value.

**5.** All enum constants must be unique in their scope. For example, the following program fails in compilation.

|  |
| --- |
| enum state  {working, failed};  enum result {failed, passed};    int main()  { return 0; } |

Run on IDE

Output:

Compile Error: 'failed' has a previous declaration as 'state failed'

**Exercise:**  
Predict the output of following C programs

Program 1:

|  |
| --- |
| #include <stdio.h>  enum day {sunday = 1, tuesday, wednesday, thursday, friday, saturday};    int main()  {      enum day d = thursday;      printf("The day number stored in d is %d", d);      return 0;  } |

Run on IDE

Program 2:

|  |
| --- |
| #include <stdio.h>  enum State {WORKING = 0, FAILED, FREEZED};  enum State currState = 2;    enum State FindState() {      return currState;  }    int main() {     (FindState() == WORKING)? printf("WORKING"): printf("NOT WORKING");     return 0;  } |

Run on IDE

**Enum vs Macro**  
We can also use macros to define names constants. For example we can define ‘Working’ and ‘Failed’ using following macro.

|  |
| --- |
| #define Working 0  #define Failed 1  #define Freezed 2 |

There are multiple advantages of using enum over macro when many related named constants have integral values.  
a) Enums follow scope rules.

b) Enum variables are automatically assigned values. Following is simpler

|  |
| --- |
| enum state  {Working, Failed, Freezed}; |

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

Extra

## Enumerated Type Declaration

When you create an enumerated type, only blueprint for the variable is created. Here's how you can create variables of enum type.

enum boolean { false, true };

enum boolean check;

Here, a variable check of type enum boolean is created.

Here is another way to declare same check variable using different syntax.

enum boolean

{

false, true

} check;

### Example: Enumeration Type

#include <stdio.h>

enum week { sunday, monday, tuesday, wednesday, thursday, friday, saturday };

int main()

{

enum week today;

today = wednesday;

printf("Day %d",today+1);

return 0;

}

**Output**

Day 4

## Why enums are used in C programming?

Enum variable takes only one value out of many possible values. Example to demonstrate it,

#include <stdio.h>

enum suit {

club = 0,

diamonds = 10,

hearts = 20,

spades = 3

} card;

int main()

{

card = club;

printf("Size of enum variable = %d bytes", sizeof(card));

return 0;

}

**Output**

Size of enum variable = 4 bytes

It's because the size of an integer is 4 bytes.

This makes enum a good choice to work with flags.

You can accomplish the same task using [structures](https://www.programiz.com/c-programming/c-structures). However, working with enums gives you efficiency along with flexibility.

### How to use enums for flags?

Let us take an example,

enum designFlags {

ITALICS = 1,

BOLD = 2,

UNDERLINE = 4

} button;

Suppose you are designing a button for Windows application. You can set flags ITALICS, BOLD and UNDERLINE to work with text.

There is a reason why all the integral constants are power of 2 in above pseudocode.

// In binary

ITALICS = 00000001

BOLD = 00000010

UNDERLINE = 00000100

Since, the integral constants are power of 2, you can combine two or more flags at once without overlapping using [bitwise OR | operator](https://www.programiz.com/c-programming/bitwise-operators#or). This allows you to choose two or more flags at once. For example,

#include <stdio.h>

enum designFlags {

BOLD = 1,

ITALICS = 2,

UNDERLINE = 4

};

int main() {

int myDesign = BOLD | UNDERLINE;

// 00000001

// | 00000100

// \_\_\_\_\_\_\_\_\_\_\_

// 00000101

printf("%d", myDesign);

return 0;

}

Output

5

When the output is 5, you always know that bold and underline is used.

Also, you can add flag to your requirements.

if (myDesign & ITALICS) {

// code for italics

}

Here, we have added italics to our design. Note, only code for italics is written inside if statement.

You can accomplish almost anything in C programming without using enumerations. However, they can be pretty handy in certain situations. That's what differentiates good programmers from great programmers.

* 1. Constant and Volatile Qualifiers

## Const (incremented, or decremented are not possible on this variable)

* const is used with a datatype declaration or definition to specify an unchanging value
* Examples:
  + const int five = 5;
  + const double pi = 3.141593;
* const objects may not be changed
  + The following are illegal:
  + const int five = 5;
  + const double pi = 3.141593;
  + pi = 3.2;
  + five = 6;

## volatile

* volatile specifies a variable whose value may be changed by processes outside the current program
* One example of a volatile object might be a buffer used to exchange data with an external device:
* int
* check\_iobuf(void)
* {
* volatile int iobuf;
* int val;
* while (iobuf == 0) {
* }
* val = iobuf;
* iobuf = 0;
* return(val);
* }
* if iobuf had not been declared volatile, the compiler would notice that nothing happens inside the loop and thus eliminate the loop
* const and volatile can be used together
* An input-only buffer for an external device could be declared as
* const volatile (or volatile const, order is not important) to make sure the compiler knows that the variable should not be changed (because it is input-only) and that its value may be altered by processes other than the current program
* In C, const and volatile are type qualifiers and these two are independent.

1. Basically, const means that the value isn’t modifiable by the program.
2. And volatile means that the value is subject to sudden change (possibly from outside the program).

* In fact, C standard mentions an example of valid declaration which is both const and volatile. The example is
* “extern const volatile int real\_time\_clock;”
* where real\_time\_clock may be modifiable by hardware, but cannot be assigned to, incremented, or decremented.
* So we should already treat const and volatile separately. Besides, these type qualifier applies for struct, union, enum and typedef as well.

|  |  |
| --- | --- |
|  | 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.  =================================================================== |

* 1. What is const key word in c

(what is constant Qualifier (variable))

* + 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;  } |

Run on IDE

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; |

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; |

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’

* 1. [**variables and constants**](https://www.codingunit.com/c-tutorial-variables-and-constants)

## Variables

In programming, a variable is a container (storage area) to hold data.

To indicate the storage area, each variable should be given a unique name ([identifier](https://www.programiz.com/c-programming/c-keywords-identifier)). Variable names are just the symbolic representation of a memory location. For example:

int playerScore = 95;

Here, playerScore is a variable of integer type. The variable is assigned value: 95.

The value of a variable can be changed, hence the name 'variable'.

In C programming, you have to declare a variable before you can use it.

### Rules for naming a variable in C

1. A variable name can have letters (both uppercase and lowercase letters), digits and underscore only.
2. The first letter of a variable should be either a letter or an underscore. However, it is discouraged to start variable name with an underscore. It is because variable name that starts with an underscore can conflict with system name and may cause error.
3. There is no rule on how long a variable can be. However, only the first 31 characters of a variable are checked by the compiler. So, the first 31 letters of two variables in a program should be different.

C is a strongly typed language. What this means it that, the type of a variable cannot be changed.

Visit this page to learn more about [different types of data a variable can store](https://www.programiz.com/c-programming/c-data-types).

## Constants/Literals

A constant is a value or an identifier whose value cannot be altered in a program. For example: 1, 2.5, "C programming is easy", etc.

As mentioned, an identifier also can be defined as a constant.

const double PI = 3.14

Here, PI is a constant. Basically what it means is that, PI and 3.14 is same for this program.

Below are the different types of constants you can use in C.

### 1. Integer constants

An integer constant is a numeric constant (associated with number) without any fractional or exponential part. There are three types of integer constants in C programming:

* decimal constant(base 10)
* octal constant(base 8)
* hexadecimal constant(base 16)

For example:

Decimal constants: 0, -9, 22 etc

Octal constants: 021, 077, 033 etc

Hexadecimal constants: 0x7f, 0x2a, 0x521 etc

In C programming, octal constant starts with a 0 and hexadecimal constant starts with a 0x.

### 2. Floating-point constants

A floating point constant is a numeric constant that has either a fractional form or an exponent form. For example:

-2.0

0.0000234

-0.22E-5

**Note:**E-5 = 10-5

### 3. Character constants

A character constant is a constant which uses single quotation around characters. For example: 'a', 'l', 'm', 'F'

### 4. Escape Sequences

Sometimes, it is necessary to use characters which cannot be typed or has special meaning in C programming. For example: newline(enter), tab, question mark etc. In order to use these characters, escape sequence is used.

For example: \n is used for newline. The backslash ( \ ) causes "escape" from the normal way the characters are interpreted by the compiler.

| Escape Sequences | |
| --- | --- |
| Escape Sequences | Character |
| \b | Backspace |
| \f | Form feed |
| \n | Newline |
| \r | Return |
| \t | Horizontal tab |
| \v | Vertical tab |
| \\ | Backslash |
| \' | Single quotation mark |
| \" | Double quotation mark |
| \? | Question mark |
| \0 | Null character |

### 5. String constants

String constants are the constants which are enclosed in a pair of double-quote marks. For example:

"good" //string constant

"" //null string constant

" " //string constant of six white space

"x" //string constant having single character.

"Earth is round\n" //prints string with newline

### 6. Enumeration constants

Keyword enum is used to define enumeration types. For example:

enum color {yellow, green, black, white};

Here, color is a variable and yellow, green, black and white are the enumeration constants having value 0, 1, 2 and 3 respectively.

# C – Constant

#### [PREV](http://fresh2refresh.com/c/c-tokens-identifiers-keywords/)     [NEXT](http://fresh2refresh.com/c/c-variables/)

* C Constants are also like normal variables. But, only difference is, their values can not be modified by the program once they are defined.
* Constants refer to fixed values. They are also called as literals
* Constants may be belonging to any of the data type.
* Syntax:

const data\_type variable\_name; (or) const data\_type \*variable\_name;

#### **TYPES OF C CONSTANT:**

1. Integer constants
2. Real or Floating point constants
3. Octal & Hexadecimal constants
4. Character constants
5. String constants
6. Backslash character constants

|  |  |
| --- | --- |
| **Constant type** | **data type (Example)** |
| Integer constants | int (53, 762, -478 etc ) unsigned int (5000u, 1000U etc) long int, long long int (483,647 2,147,483,680) |
| Real or Floating point constants | float (10.456789) doule (600.123456789) |
| Octal constant | int (Example: 013 /\*starts with 0 \*/) |
| Hexadecimal constant | int (Example: 0x90 /\*starts with 0x\*/) |
| character constants | char (Example: ‘A’, ‘B’, ‘C’) |
| string constants | char (Example: “ABCD”, “Hai”) |

#### **RULES FOR CONSTRUCTING C CONSTANT:**

#### **1. INTEGER CONSTANTS IN C:**

* An integer constant must have at least one digit.
* It must not have a decimal point.
* It can either be positive or negative.
* No commas or blanks are allowed within an integer constant.
* If no sign precedes an integer constant, it is assumed to be positive.
* The allowable range for integer constants is -32768 to 32767.

#### **2. REAL CONSTANTS IN C:**

* A real constant must have at least one digit
* It must have a decimal point
* It could be either positive or negative
* If no sign precedes an integer constant, it is assumed to be positive.
* No commas or blanks are allowed within a real constant.

#### **3. CHARACTER AND STRING CONSTANTS IN C:**

* A character constant is a single alphabet, a single digit or a single special symbol enclosed within single quotes.
* The maximum length of a character constant is 1 character.
* String constants are  enclosed within double quotes.
* 4. BACKSLASH CHARACTER CONSTANTS IN C:
* There are some characters which have special meaning in C language.
* They should be preceded by backslash symbol to make use of special function of them.
* Given below is the list of special characters and their purpose.

|  |  |
| --- | --- |
| **Backslash\_character** | **Meaning** |
| \b | Backspace |
| \f | Form feed |
| \n | New line |
| \r | Carriage return |
| \t | Horizontal tab |
| \” | Double quote |
| \’ | Single quote |
| \\ | Backslash |
| \v | Vertical tab |
| \a | Alert or bell |
| \? | Question mark |
| \N | Octal constant (N is an octal constant) |
| \XN | Hexadecimal constant (N – hex.dcml cnst) |

#### **HOW TO USE CONSTANTS IN A C PROGRAM?**

* We can define constants in a C program in the following ways.
* By “const” keyword
* By “#define” preprocessor directive
* Please note that when you try to change constant values after defining in C program, it will through error.

#### **1. EXAMPLE PROGRAM USING CONST KEYWORD IN C:**

C



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | #include <stdio.h>  void main()  {  const int height = 100; /\*int constant\*/  const float number = 3.14; /\*Real constant\*/  const char letter = 'A'; /\*char constant\*/  const char letter\_sequence[10] = "ABC"; /\*string constant\*/  const char backslash\_char = '\?'; /\*special char cnst\*/  printf("value of height :%d \n", height );  printf("value of number : %f \n", number );  printf("value of letter : %c \n", letter );  printf("value of letter\_sequence : %s \n", letter\_sequence);  printf("value of backslash\_char : %c \n", backslash\_char);  } |

#### **OUTPUT:**

|  |
| --- |
| value of height : 100 value of number : 3.140000 value of letter : A value of letter\_sequence : ABC value of backslash\_char : ? |

#### **2. EXAMPLE PROGRAM USING #DEFINE PREPROCESSOR DIRECTIVE IN C:**

C



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | #include <stdio.h>  #define height 100  #define number 3.14  #define letter 'A'  #define letter\_sequence "ABC"  #define backslash\_char '\?'  void main()  {  printf("value of height : %d \n", height );  printf("value of number : %f \n", number );  printf("value of letter : %c \n", letter );  printf("value of letter\_sequence : %s \n",letter\_sequence);  printf("value of backslash\_char : %c \n",backslash\_char);  } |

**Output:**

|  |
| --- |
| value of height : 100 value of number : 3.140000 value of letter : A value of letter\_sequence : ABC value of backslash\_char : ? |

* 1. Command Line Arguments

It is possible to pass some values from the command line to your C programs when they are executed. These values are called **command line arguments**and many times they are important for your program especially when you want to control your program from outside instead of hard coding those values inside the code.

The command line arguments are handled using main() function arguments where **argc** refers to the number of arguments passed, and **argv[]** is a pointer array which points to each argument passed to the program. Following is a simple example which checks if there is any argument supplied from the command line and take action accordingly −

#include <stdio.h>

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

if( argc == 2 ) {

printf("The argument supplied is %s\n", argv[1]);

}

else if( argc > 2 ) {

printf("Too many arguments supplied.\n");

}

else {

printf("One argument expected.\n");

}

}

When the above code is compiled and executed with single argument, it produces the following result.

$./a.out testing

The argument supplied is testing

When the above code is compiled and executed with a two arguments, it produces the following result.

$./a.out testing1 testing2

Too many arguments supplied.

When the above code is compiled and executed without passing any argument, it produces the following result.

$./a.out

One argument expected

It should be noted that **argv[0]** holds the name of the program itself and **argv[1]** is a pointer to the first command line argument supplied, and \*argv[n] is the last argument. If no arguments are supplied, argc will be one, and if you pass one argument then **argc** is set at 2.

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 ''. Let us re-write above example once again where we will print program name and we also pass a command line argument by putting inside double quotes −

#include <stdio.h>

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

printf("Program name %s\n", argv[0]);

if( argc == 2 ) {

printf("The argument supplied is %s\n", argv[1]);

}

else if( argc > 2 ) {

printf("Too many arguments supplied.\n");

}

else {

printf("One argument expected.\n");

}

}

When the above code is compiled and executed with a single argument separated by space but inside double quotes, it produces the following result.

$./a.out "testing1 testing2"

Progranm name ./a.out

The argument supplied is testing1 testing2

# Accepting command line arguments in C using argc and argv

In C it is possible to accept command line arguments. Command-line arguments are given after the name of a program in command-line operating systems like DOS or Linux, and are passed in to the program from the operating system. To use command line arguments in your program, you must first understand the full declaration of the main function, which previously has accepted no arguments. In fact, main can actually accept two arguments: one argument is number of command line arguments, and the other argument is a full list of all of the command line arguments. 

The full declaration of main looks like this:

int main ( int argc, char \*argv[] )

The integer, argc is the argument count. It is the number of arguments passed into the program from the command line, including the name of the program.   
  
The array of character pointers is the listing of all the arguments. argv[0] is the name of the program, or an empty string if the name is not available. After that, every element number less than argc is a command line argument. You can use each argv element just like a string, or use argv as a two dimensional array. argv[argc] is a null pointer.   
  
How could this be used? Almost any program that wants its parameters to be set when it is executed would use this. One common use is to write a function that takes the name of a file and outputs the entire text of it onto the screen.

#include <stdio.h>

int main ( int argc, char \*argv[] )

{

if ( argc != 2 ) /\* argc should be 2 for correct execution \*/

{

/\* We print argv[0] assuming it is the program name \*/

printf( "usage: %s filename", argv[0] );

}

else

{

// We assume argv[1] is a filename to open

FILE \*file = fopen( argv[1], "r" );

/\* fopen returns 0, the NULL pointer, on failure \*/

if ( file == 0 )

{

printf( "Could not open file\n" );

}

else

{

int x;

/\* read one character at a time from file, stopping at EOF, which

indicates the end of the file. Note that the idiom of "assign

to a variable, check the value" used below works because

the assignment statement evaluates to the value assigned. \*/

while ( ( x = fgetc( file ) ) != EOF )

{

printf( "%c", x );

}

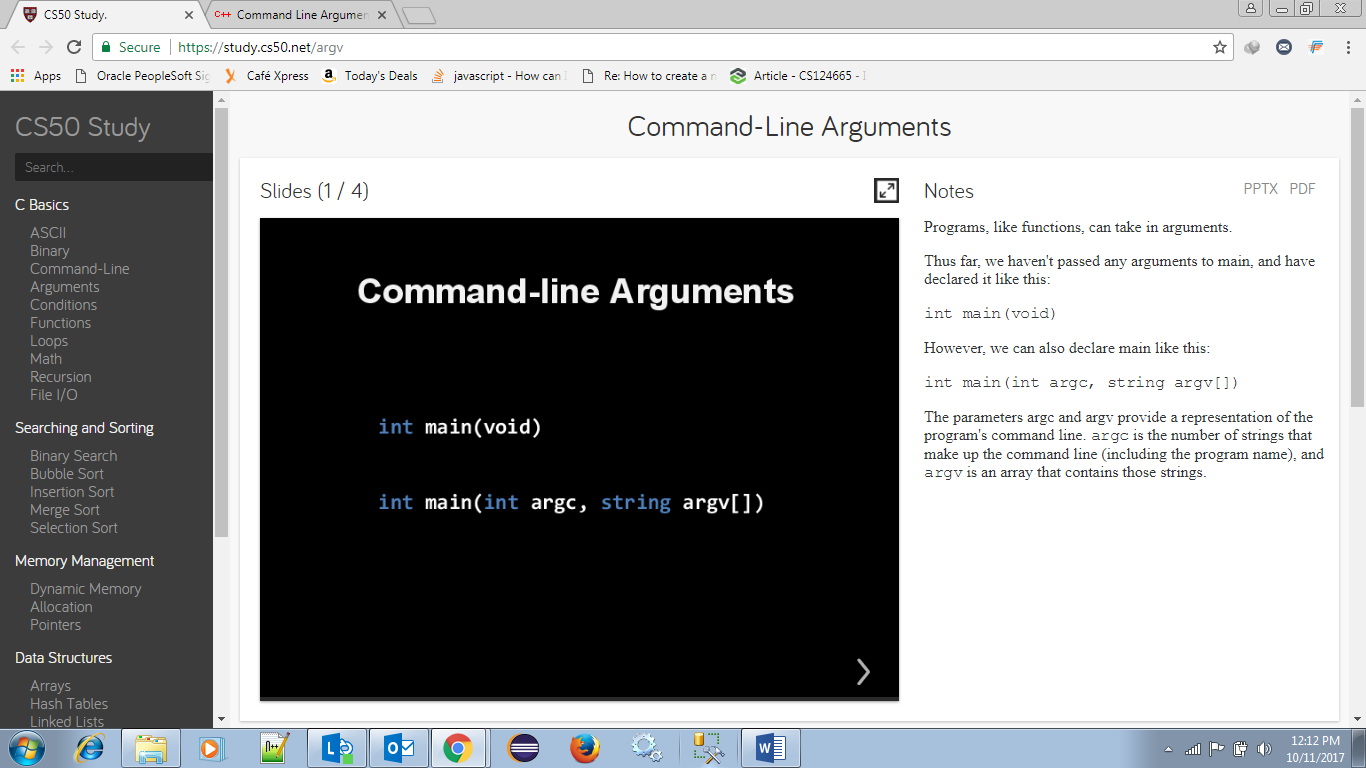
fclose( file );

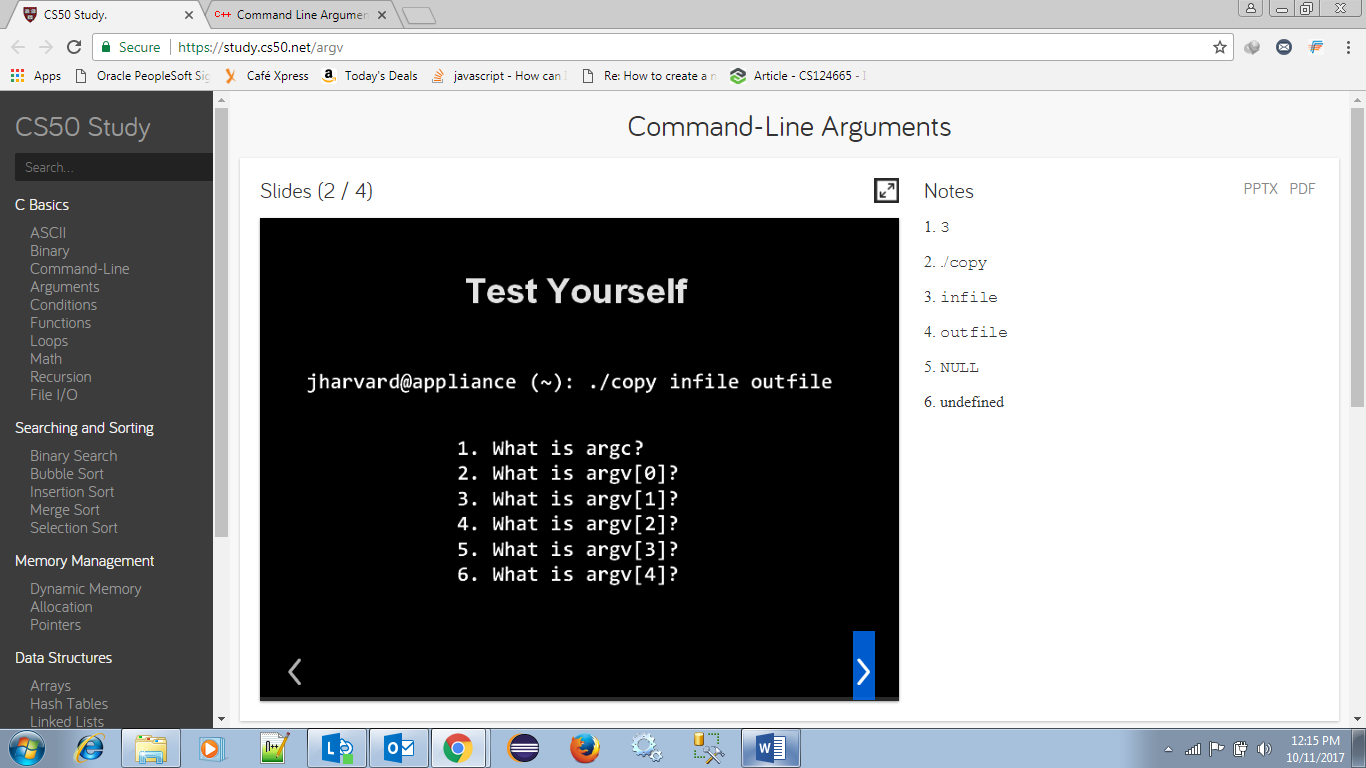
}

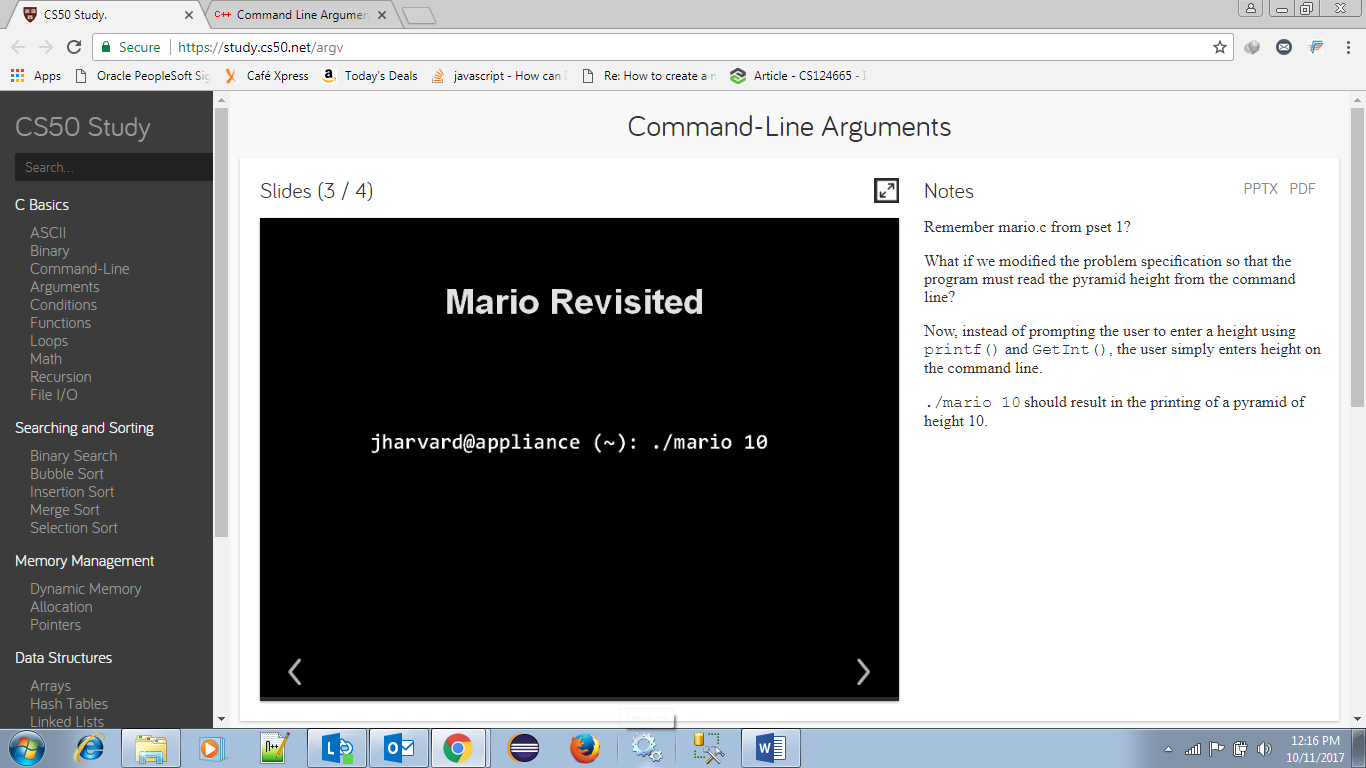
}

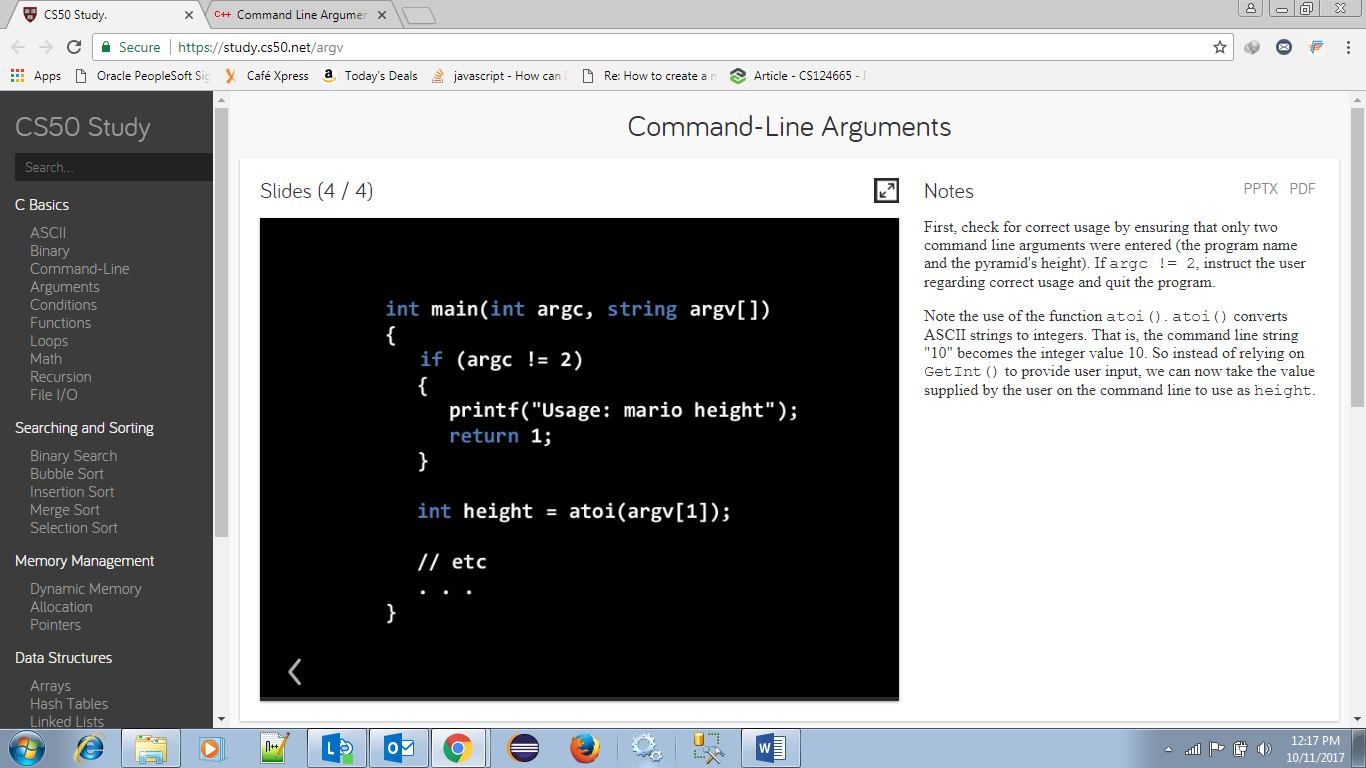
}

This program is fairly short, but it incorporates the full version of main and even performs a useful function. It first checks to ensure the user added the second argument, theoretically a file name. The program then checks to see if the file is valid by trying to open it. This is a standard operation, and if it results in the file being opened, then the return value of fopen will be a valid FILE\*; otherwise, it will be 0, the NULL pointer. After that, we just execute a loop to print out one character at a time from the file. The code is self-explanatory, but is littered with comments;









* 1. What is Memory Leak? How can we avoid?

Memory leak occurs when programmers create a memory in heap and forget to delete it.  
Memory leaks are particularly serious issues for programs like daemons and servers which by definition never terminate.

|  |
| --- |
| /\* Function with memory leak \*/  #include <stdlib.h>    void f()  {     int \*ptr = (int \*) malloc(sizeof(int));       /\* Do some work \*/       return; /\* Return without freeing ptr\*/  } |

* To avoid memory leaks, memory allocated on heap should always be freed when no longer needed.

|  |
| --- |
| /\* Function without memory leak \*/  #include <stdlib.h>;    void f()  {     int \*ptr = (int \*) malloc(sizeof(int));       /\* Do some work \*/       free(ptr);     return;  }  \*Memory Leak: A memory leak occurs when a piece (or pieces) of memory that was previously allocated by a programmer is not properly deallocated by the programmer. Even though that memory is no longer in use by the program, it is still “reserved”,  and that piece of memory can not be used by the program until it is properly deallocated by the programmer.  That’s why it’s called a memory leak   ex: To avoid memory leaks, memory allocated on heap should always be freed when no longer needed.          /\* Function without memory leak \*/             #include <stdlib.h>;               void f()             {                    int \*ptr = (int \*) malloc(sizeof(int));                      /\* Do some work \*/                      free(ptr);                   return;             }  Memory leak occurs when programmers create a memory in heap and forget to delete it. Memory leaks are particularly serious issues for programs like daemons and servers which by definition never terminate. |

#### Sabith

* Jul 10th, 2006

In computer science, a memory leak is a particular kind of unintentional memory consumption by a computer program where the program fails to release memory when no longer needed. The term is meant as a humorous misnomer, since memory is not physically lost from the computer, but rather becomes claimed but ignored due to program logic flaws. When we are allocating memory using new or malloc and then failing to deallocate the memory after its use within the application's life span, then the memory allocated earlier will become unusable for the application, there by making it a leak in the memory.

  Was this answer useful?  [Yes](http://www.geekinterview.com/login.html?redirect=/question_details/31226)

Reply

#### Sabith

* Jul 10th, 2006

A memory leak can be avoided by making sure that whatever memory has been dynamically allocated will be cleared after the use of the same. for example   
int main()   
{ char \*myCharData[20];   
for (int nLoop =0;nLoop < 20; ++nLoop) { myCharData[nLoop ] = new char[256];   
strcpy(myCharData[nLoop],"SABITH");   
.......   
}   
.........................   
/\*Some manipulations here using myCharData\*/   
/\*Now here we have to clear the data. The place can vary according to ur program. This being a simple program,u can clear at the end\*/   
for(int nLoop =0;nLoop < 20; ++nLoop)   
{   
delete[] myCharData[nLoop ];   
}   
return 0;   
}

  Was this answer useful?  [Yes](http://www.geekinterview.com/login.html?redirect=/question_details/31226)

Reply

#### Guest

* Jul 17th, 2006

When dynamic memory allocation is used extensively, memory leak may occur.It happens when first memory block is not deleted . However, the address is lost because the pointer contains the address of second block.The memory leak is unintentional occupied memory. It can de avoided by using delete statement.

example:

float \*ptr=new float

\*ptr=6.0;//Access first block

'Use delete statement --"delete ptr";Otherwise it will cause memory leak

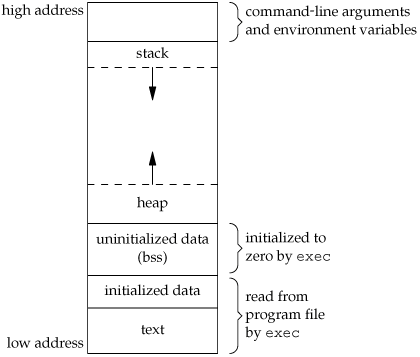
ptr=new float

\*ptr=7.0//Acess second block

11 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

[](http://www.geeksforgeeks.org/wp-content/uploads/Memory-Layout.gif)  
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.

1. **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

1. **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.

1. **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.

1. **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.

# 12. C – Function Pointer with examples

BY CHAITANYA SINGH | FILED UNDER: [C-PROGRAMMING](https://beginnersbook.com/category/c-programming/)

In [C programming language](https://beginnersbook.com/2014/01/c-tutorial-for-beginners-with-examples/), we can have a concept of Pointer to a function known as **function pointer in C**. In this tutorial, we will learn how to declare a function pointer and how to call a function using this pointer. To understand this concept, you should have the basic knowledge of [Functions](https://beginnersbook.com/2014/01/c-functions-examples/) and [Pointers in C](https://beginnersbook.com/2014/01/c-pointers/).

## How to declare a function pointer?

function\_return\_type(\*Pointer\_name)(function argument list)

For example:

double  (\*p2f)(double, char)

Here double is a return type of function, p2f is name of the function pointer and (double, char) is an argument list of this function. Which means the first argument of this function is of double type and the second argument is char type.

Lets understand this with the help of an example:

Here we have a function sum that calculates the sum of two numbers and returns the sum. We have created a pointer f2p that points to this function, we are invoking the function using this function pointer f2p.

int sum (int num1, int num2)

{

return num1+num2;

}

int main()

{

/\* The following two lines can also be written in a single

\* statement like this: void (\*fun\_ptr)(int) = &fun;

\*/

int (\*f2p) (int, int);

f2p = sum;

//Calling function using function pointer

int op1 = f2p(10, 13);

//Calling function in normal way using function name

int op2 = sum(10, 13);

printf("Output1: Call using function pointer: %d",op1);

printf("\nOutput2: Call using function name: %d", op2);

return 0;

}

**Output:**

Output1: Call using function pointer: 23

Output2: Call using function name: 23

**Some points regarding function pointer:**  
1. As mentioned in the comments, you can declare a function pointer and assign a function to it in a single statement like this:

void (\*fun\_ptr)(int) = &fun;

2. You can even remove the ampersand from this statement because a function name alone represents the function address. This means the above statement can also be written like this:

void (\*fun\_ptr)(int) = fun;

# Function Pointer in C

In C, like [normal data pointers](http://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;  } |

Run 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?](http://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;  } |

Run 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;  } |

Run 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;  } |

Run 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;  } |

Run 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;  } |

Run 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](http://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.

**Related Article:**[Pointers in C and C++ | Set 1 (Introduction, Arithmetic and Array)](http://www.geeksforgeeks.org/pointers-in-c-and-c-set-1-introduction-arithmetic-and-array/)

**References:**  
<http://www.cs.cmu.edu/~ab/15-123S11/AnnotatedNotes/Lecture14.pdf>

<http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-087-practical-programming-in-c-january-iap-2010/lecture-notes/MIT6_087IAP10_lec08.pdf>

<http://www.cs.cmu.edu/~guna/15-123S11/Lectures/Lecture14.pdf>

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.

# 13 C – Preprocessor directives

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

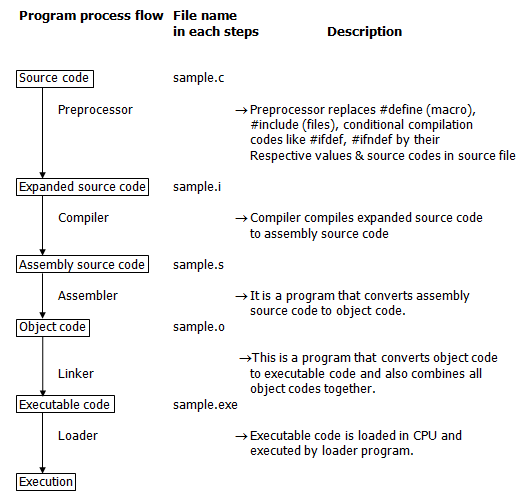
#### **C PREPROCESSOR DIRECTIVES:**

* 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. |

#### **DO YOU KNOW DIFFERENCE BETWEEN COMPILERS VS INTERPRETERS IN C LANGUAGE?**

|  |  |
| --- | --- |
| **Compilers** | **Interpreters** |
| Compiler reads the entire source code of the program and converts it into binary code. This process is called compilation.  Binary code is also referred as machine code, executable, and object code. | Interpreter reads the program source code one line at a time and executing that line. This process is called interpretation. |
| Program speed is fast. | Program speed is slow. |
| One time execution. Example: C, C++ | Interpretation occurs at every line of the program. Example: BASIC |

#### **KEY POINTS TO REMEMBER:**

1. Source program is converted into executable code through different processes like precompilation, compilation, assembling and linking.
2. Local variables uses stack memory.
3. Dynamic memory allocation functions use the heap memory.

#### **EXAMPLE PROGRAM FOR #DEFINE, #INCLUDE PREPROCESSORS IN C LANGUAGE:**

* #define**–**This macro defines constant value and can be any of the basic data types.
* #include <file\_name>**–**The source code of the file “file\_name” is included in the main C program where “#include <file\_name>” is mentioned.

C



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | #include <stdio.h>    #define height 100  #define number 3.14  #define letter 'A'  #define letter\_sequence "ABC"  #define backslash\_char '\?'    void main()  {     printf("value of height    : %d \n", height );     printf("value of number : %f \n", number );     printf("value of letter : %c \n", letter );     printf("value of letter\_sequence : %s \n", letter\_sequence);     printf("value of backslash\_char  : %c \n", backslash\_char);    } |

#### **OUTPUT:**

|  |
| --- |
| value of height : 100  value of number : 3.140000  value of letter : A  value of letter\_sequence : ABC  value of backslash\_char : ? |

#### **EXAMPLE PROGRAM FOR CONDITIONAL COMPILATION DIRECTIVES:**

#### **A) EXAMPLE PROGRAM FOR #IFDEF, #ELSE AND #ENDIF IN C:**

* “#ifdef” directive checks whether particular macro is defined or not. If it is defined, “If” clause statements are included in source file.
* Otherwise, “else” clause statements are included in source file for compilation and execution.

C



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | #include <stdio.h>  #define RAJU 100    int main()  {     #ifdef RAJU     printf("RAJU is defined. So, this line will be added in " \            "this C file\n");     #else     printf("RAJU is not defined\n");     #endif     return 0;  } |

#### **OUTPUT:**

|  |
| --- |
| RAJU is defined. So, this line will be added in this C file |

#### **B) EXAMPLE PROGRAM FOR #IFNDEF AND #ENDIF IN C:**

* #ifndef exactly acts as reverse as #ifdef directive. If particular macro is not defined, “If” clause statements are included in source file.
* Otherwise, else clause statements are included in source file for compilation and execution.

C



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | #include <stdio.h>  #define RAJU 100  int main()  {     #ifndef SELVA     {        printf("SELVA is not defined. So, now we are going to " \               "define here\n");        #define SELVA 300     }     #else     printf("SELVA is already defined in the program”);       #endif     return 0;    } |

#### **OUTPUT:**

|  |
| --- |
| SELVA is not defined. So, now we are going to define here |

#### **C) EXAMPLE PROGRAM FOR #IF, #ELSE AND #ENDIF IN C:**

* “If” clause statement is included in source file if given condition is true.
* Otherwise, else clause statement is included in source file for compilation and execution.

C



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | #include <stdio.h>  #define a 100  int main()  {     #if (a==100)     printf("This line will be added in this C file since " \            "a \= 100\n");     #else     printf("This line will be added in this C file since " \            "a is not equal to 100\n");     #endif     return 0;  } |

#### **OUTPUT:**

|  |
| --- |
| This line will be added in this C file since a = 100 |

#### **EXAMPLE PROGRAM FOR UNDEF IN C LANGUAGE:**

This directive undefines existing macro in the program.

C



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | #include <stdio.h>    #define height 100  void main()  {     printf("First defined value for height    : %d\n",height);     #undef height          // undefining variable     #define height 600     // redefining the same for new value     printf("value of height after undef \& redefine:%d",height);  } |

#### **OUTPUT:**

|  |
| --- |
| First defined value for height : 100  value of height after undef & redefine : 600 |

#### **EXAMPLE PROGRAM FOR PRAGMA IN C LANGUAGE:**

Pragma is used to call a function before and after main function in a C program.

C



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | #include <stdio.h>    void function1( );  void function2( );    #pragma startup function1  #pragma exit function2    int main( )  {     printf ( "\n Now we are in main function" ) ;     return 0;  }    void function1( )  {     printf("\nFunction1 is called before main function call");  }    void function2( )  {     printf ( "\nFunction2 is called just before end of " \              "main function" ) ;"  } |

#### **OUTPUT:**

|  |
| --- |
| Function1 is called before main function call  Now we are in main function  Function2 is called just before end of main function |

#### **MORE ON PRAGMA DIRECTIVE IN C LANGUAGE:**

|  |  |
| --- | --- |
| **Pragma command** | **Description** |
| #Pragma startup <function\_name\_1> | This directive executes function named “function\_name\_1” before |
| #Pragma exit <function\_name\_2> | This directive executes function named “function\_name\_2” just before termination of the program. |
| #pragma warn – rvl | If function doesn’t return a value, then warnings are suppressed by this directive while compiling. |
| #pragma warn – par | If function doesn’t use passed function parameter , then warnings are suppressed |
| #pragma warn – rch | If a non reachable code is written inside a program, such warnings are suppressed by this directive. |

# 14. #Pragma Directives and the \_\_Pragma Keyword

**Define #pragma statements.**

**The #pragma Directives are used to turn ON or OFF certain features**. They vary from compiler to compiler.

**When we are going to compile the code, the part of the code written under the #pragma will not compile until unless we will define that**Examples of pragmas are:

#pragma startup // you can use this to execute a function at startup of a program  
#pragma exit // you can use this to execute a function at exiting of a program

# #pragma

#pragma *compiler specific extension*

The pragma directive is used to access compiler-specific preprocessor extensions. A common use of #pragma is the #pragma once directive, which asks the compiler to include a header file only a single time, no matter how many times it has been imported:

#pragma once

// header file code

In this example, using #pragma once is equivalent to an [include guard](https://www.cprogramming.com/tutorial/cpreprocessor.html) that prevents the file from being processed multiple times.

#ifndef \_FILE\_NAME\_H\_

#define \_FILE\_NAME\_H\_

/\* code \*/

#endif // #ifndef \_FILE\_NAME\_H\_

#pragma once is available on many major compilers, including Clang, [GCC](https://www.cprogramming.com/g++.html), the Intel C++ compiler and [MSVC](https://www.cprogramming.com/visual.html).

The #pragma directive can also be used for other compiler-specific purposes. #pragma is commonly used to suppress warnings. For example, in MSVC

#pragma warning (disable : 4018 )

Can be used to disable warning 4018, warning of signed/unsigned mismatch. While [you should be reluctant to suppress warnings](https://www.cprogramming.com/tutorial/compiler_warnings.html) sometimes it is necessary.

For more uses of the #pragma directive, consult your compiler's documentation.

# Bitwise Operators in C

The following table lists the Bitwise operators supported by C. Assume variable 'A' holds 60 and variable 'B' holds 13, then −

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| & | Binary AND Operator copies a bit to the result if it exists in both operands. | (A & B) = 12 i.e., 0000 1100 |
| ∣ | Binary OR Operator copies a bit if it exists in either operand. | (A ∣ B) = 61 i.e., 0011 1101 |
| ^ | Binary XOR Operator copies the bit if it is set in one operand but not both. | (A ^ B) = 49 i.e., 0011 0001 |
| ~ | Binary Ones Complement Operator is unary and has the effect of 'flipping' bits. | (~A ) = -61 i.e., 1100 0011 in 2's complement form. |
| << | Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand. | A << 2 = 240 i.e., 1111 0000 |
| >> | Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand. | A >> 2 = 15 i.e., 0000 1111 |

## **Example**

Try the following example to understand all the bitwise operators available in C −

#include <stdio.h>

main() {

unsigned int a = 60; /\* 60 = 0011 1100 \*/

unsigned int b = 13; /\* 13 = 0000 1101 \*/

int c = 0;

c = a & b; /\* 12 = 0000 1100 \*/

printf("Line 1 - Value of c is %d\n", c );

c = a | b; /\* 61 = 0011 1101 \*/

printf("Line 2 - Value of c is %d\n", c );

c = a ^ b; /\* 49 = 0011 0001 \*/

printf("Line 3 - Value of c is %d\n", c );

c = ~a; /\*-61 = 1100 0011 \*/

printf("Line 4 - Value of c is %d\n", c );

c = a << 2; /\* 240 = 1111 0000 \*/

printf("Line 5 - Value of c is %d\n", c );

c = a >> 2; /\* 15 = 0000 1111 \*/

printf("Line 6 - Value of c is %d\n", c );

}

When you compile and execute the above program, it produces the following result −

Line 1 - Value of c is 12

Line 2 - Value of c is 61

Line 3 - Value of c is 49

Line 4 - Value of c is -61

Line 5 - Value of c is 240

Line 6 - Value of c is 15

# Interesting Facts about Bitwise Operators in C

In C, following 6 operators are bitwise operators (work at bit-level)

**& (bitwise AND)** Takes two numbers as operand and does AND on every bit of two numbers. The result of AND is 1 only if both bits are 1.

**| (bitwise OR)** Takes two numbers as operand and does OR on every bit of two numbers. The result of OR is 1 any of the two bits is 1.

**^ (bitwise XOR)** Takes two numbers as operand and does XOR on every bit of two numbers. The result of XOR is 1 if the two bits are different.

**<< (left shift)** Takes two numbers, left shifts the bits of first operand, the second operand decides the number of places to shift.

**>> (right shift)** Takes two numbers, right shifts the bits of first operand, the second operand decides the number of places to shift.

**~ (bitwise NOT)** Takes one number and inverts all bits of it

Following is example C program.

|  |
| --- |
| /\* C Program to demonstrate use of bitwise operators \*/  #include<stdio.h>  int main()  {      unsigned char a = 5, b = 9; // a = 4(00000101), b = 8(00001001)      printf("a = %d, b = %d\n", a, b);      printf("a&b = %d\n", a&b); // The result is 00000001      printf("a|b = %d\n", a|b);  // The result is 00001101      printf("a^b = %d\n", a^b); // The result is 00001100      printf("~a = %d\n", a = ~a);   // The result is 11111010      printf("b<<1 = %d\n", b<<1);  // The result is 00010010      printf("b>>1 = %d\n", b>>1);  // The result is 00000100      return 0;  } |

Run on IDE

Output:

a = 5, b = 9

a&b = 1

a|b = 13

a^b = 12

~a = 250

b<<1 = 18

b>>1 = 4

Following are interesting facts about bitwise operators.

**1) The left shift and right shift operators should not be used for negative numbers** The result of << and >> is undefined behabiour if any of the operands is a negative number. For example results of both -1 << 1 and 1 << -1 is undefined. Also, if the number is shifted more than the size of integer, the behaviour is undefined. For example, 1 << 33 is undefined if integers are stored using 32 bits. See [this](https://www.securecoding.cert.org/confluence/display/seccode/INT34-C.+Do+not+shift+a+negative+number+of+bits+or+more+bits+than+exist+in+the+operand) for more details.

**2) The bitwise XOR operator is the most useful operator from technical interview perspective.**It is used in many problems. A simple example could be “Given a set of numbers where all elements occur even number of times except one number, find the odd occuring number” This problem can be efficiently solved by just doing XOR of all numbers.

|  |
| --- |
| // Function to return the only odd occurring element  int findOdd(int arr[], int n) {     int res = 0, i;     for (i = 0; i < n; i++)       res ^= arr[i];     return res;  }    int main(void) {     int arr[] = {12, 12, 14, 90, 14, 14, 14};     int n = sizeof(arr)/sizeof(arr[0]);     printf ("The odd occurring element is %d ", findOdd(arr, n));     return 0;  }  // Output: The odd occurring element is 90 |

Run on IDE

The following are many other interesting problems which can be used using XOR operator.  
[Find the Missing Number](http://www.geeksforgeeks.org/find-the-missing-number/), [swap two numbers without using a temporary variable](http://www.geeksforgeeks.org/swap-two-numbers-without-using-temporary-variable/), [A Memory Efficient Doubly Linked List](http://www.geeksforgeeks.org/xor-linked-list-a-memory-efficient-doubly-linked-list-set-1/), and [Find the two non-repeating elements](http://www.geeksforgeeks.org/find-two-non-repeating-elements-in-an-array-of-repeating-elements/). There are many more (See [this](http://www.geeksforgeeks.org/find-the-two-numbers-with-odd-occurences-in-an-unsorted-array/), [this](http://www.geeksforgeeks.org/add-two-numbers-without-using-arithmetic-operators/), [this](http://www.geeksforgeeks.org/swap-bits-in-a-given-number/), [this](http://www.geeksforgeeks.org/count-number-of-bits-to-be-flipped-to-convert-a-to-b/), [this](http://www.geeksforgeeks.org/find-the-element-that-appears-once/) and [this](http://www.geeksforgeeks.org/detect-if-two-integers-have-opposite-signs/))

**3) The bitwise operators should not be used in-place of logical operators.**  
The result of logical operators (&&, || and !) is either 0 or 1, but bitwise operators return an integer value. Also, the logical operators consider any non-zero operand as 1. For example consider the following program, the results of & and && are different for same operands.

|  |
| --- |
| int main()  {     int x = 2, y = 5;     (x & y)? printf("True ") : printf("False ");     (x && y)? printf("True ") : printf("False ");     return 0;  }  // Output: False True |

Run on IDE

**4) The left-shift and right-shift operators are equivalent to multiplication and division by 2 respectively.**  
As mentioned in point 1, it works only if numbers are positive.

|  |
| --- |
| int main()  {     int x = 19;     printf ("x << 1 = %d\n", x << 1);     printf ("x >> 1 = %d\n", x >> 1);     return 0;  }  // Output: 38 9 |

Run on IDE

**5) The & operator can be used to quickly check if a number is odd or even**

The value of expression (x & 1) would be non-zero only if x is odd, otherwise the value would be zero.

|  |
| --- |
| int main()  {     int x = 19;     (x & 1)? printf("Odd"): printf("Even");     return 0;  }  // Output: Odd |

Run on IDE

**6) The ~ operator should be used carefully**

The result of ~ operator on a small number can be a big number if result is stored in a unsigned variable. And result may be negative number if result is stored in signed variable (assuming that the negative numbers are stored in 2’s complement form where leftmost bit is the sign bit)

|  |
| --- |
| // Note that the output of following program is compiler dependent  int main()  {     unsigned int x = 1;     printf("Signed Result %d \n", ~x);     printf("Unsigned Result %ud \n", ~x);     return 0;  }  /\* Output:  Signed Result -2  Unsigned Result 4294967294d \*/ 15. Data Structure alignment and Padding |

Data structure alignment is the way data is arranged and accessed in computer memory. Data alignment and Data structure padding are two different issues but are related to each other and together known as Data Structure alignment.

**Data alignment**: Data alignment means putting the data in memory at address equal to some multiple of the word size. This increases the performance of system due to the way the CPU handles memory.

**Data Structure Padding**: Now, to align the data, it may be necessary to insert some extra bytes between the end of the last data structure and the start of the next data structure as the data is placed in memory as multiples of fixed word size. This insertion of extra bytes of memory to align the data is called data structure padding.

Consider the structure as shown below:

struct

{

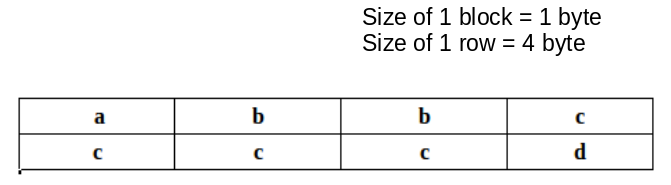
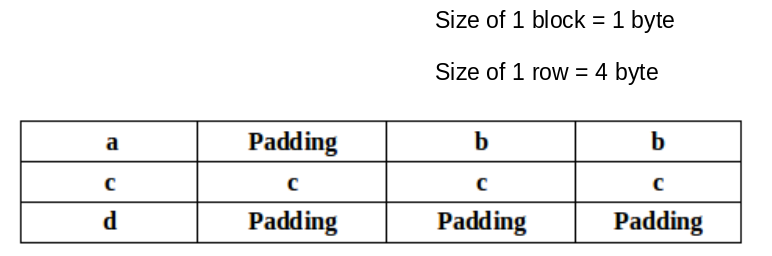
char a;

short int b;

int c;

char d;

}

Now we may think that the processor will allocate memory to this structure as shown below:  
  
The total memory allocated in this case is 8 bytes. But this never happens as the processor can access memory as fixed word size of 4 bytes. So, the integer variable c can not be allocated memory as shown above. An integer variable requires 4 bytes. The correct way of allocation of memory is shown below for this structure using padding bytes.  
  
The processor will require a total of 12 bytes for the above structure to maintain the data alignment.  
Look at the below C++ program:

|  |
| --- |
| // CPP program to test  // size of struct  #include <iostream>  using namespace std;    // first structure  struct test1  {      short s;      int i;      char c;  };    // second structure  struct test2  {      int i;      char c;      short s;  };    // driver program  int main()  {      test1 t1;      test2 t2;      cout << "size of struct test1 is " << sizeof(t1) << "\n";      cout << "size of struct test2 is " << sizeof(t2) << "\n";      return 0;  } |

Output:

size of struct test1 is 12

size of struct test2 is 8

For the first structure test1 the short variable takes 2 byte. Now the next variable is int which requires 4 bytes. So, 2 bytes padding is added after short variable. Now, the char variable requires 1 byte but memory will be accessed in word size of 4 byte so 3 byte of padding is added again. So, total 12 bytes of memory is required. We can similarly calculate the padding for the second structure also. Padding for both of the structures is shown below:

struct test1

{

short s;

// 2 bytes

// 2 padding bytes

int i;

// 4 bytes

char c;

// 1 byte

// 3 padding bytes

};

struct test2

{

int i;

// 4 bytes

char c;

// 1 byte

// 1 padding byte

short s;

// 2 bytes

};

Note :

You can minimize the size of memory allocated for a structure by sorting members by alignment.

Data structure alignment is the way data is arranged and accessed in computer memory. It consists of two separate but related issues: data alignment and [data structure padding](http://www.firmcodes.com/structure-padding-and-packing-in-c-example/). When a modern computer reads from or writes to a memory address, it will do this in word sized chunks (e.g. 4 byte chunks on a 32-bit system) or larger. Data alignment means putting the data at a memory address equal to some multiple of the word size, which increases the system’s performance due to the way the CPU handles memory. To align the data, it may be necessary to insert some meaningless bytes between the end of the last data structure and the start of the next, which is data structure padding.

In order to align the data in memory, one or more empty bytes (addresses) are inserted (or left empty) between memory addresses which are allocated for other structure members while memory allocation. This concept is called structure padding.

Architecture of a computer processor is such a way that it can read 1 word (4 byte in 32 bit processor) from memory at a time.

To make use of this advantage of processor, data are always aligned as 4 bytes package which leads to insert empty addresses between other member’s address.

Because of this structure padding concept in C, size of the structure is always not same as what we think.

|  |  |
| --- | --- |
|  | Structure packing is only done when you tell your compiler explicitly to pack the structure. Padding is what you're seeing. Your 32-bit system is padding each field to word alignment. If you had told your compiler to pack the structures, they'd be 6 and 5 bytes, respectively. Don't do that though. It's not portable and makes compilers generate much slower (and sometimes even buggy) code. |

## 3. Alignment requirements

The first thing to understand is that, on modern processors, the way your C compiler lays out basic C datatypes in memory is constrained in order to make memory accesses faster.

Storage for the basic C datatypes on an x86 or ARM processor doesn’t normally start at arbitrary byte addresses in memory. Rather, each type except char has an *alignment requirement*; chars can start on any byte address, but 2-byte shorts must start on an even address, 4-byte ints or floats must start on an address divisible by 4, and 8-byte longs or doubles must start on an address divisible by 8. Signed or unsigned makes no difference.

The jargon for this is that basic C types on x86 and ARM are *self-aligned*. Pointers, whether 32-bit (4-byte) or 64-bit (8-byte) are self-aligned too.

Self-alignment makes access faster because it facilitates generating single-instruction fetches and puts of the typed data. Without alignment constraints, on the other hand, the code might end up having to do two or more accesses spanning machine-word boundaries. Characters are a special case; they’re equally expensive from anywhere they live inside a single machine word. That’s why they don’t have a preferred alignment.

I said "on modern processors" because on some older ones forcing your C program to violate alignment rules (say, by casting an odd address into an int pointer and trying to use it) didn’t just slow your code down, it caused an illegal instruction fault. This was the behavior, for example, on Sun SPARC chips. In fact, with sufficient determination and the right (e18) hardware flag set on the processor, you can still trigger this on x86.

Also, self-alignment is not the only possible rule. Historically, some processors (especially those lacking [barrel shifters](https://en.wikipedia.org/wiki/Barrel_shifter)) have had more restrictive ones. If you do embedded systems, you might trip over one of these lurking in the underbrush. Be aware this is possible.

From when it was written at the beginning of 2014 until late 2016 this section ended with the last paragraph. During that period I’ve learned something rather reassuring from working with the source code for the reference implementation of NTP. It does packet analysis by reading packets off the wire directly into memory that the rest of the code sees as a struct, relying on the assumption of minimal self-aligned padding.

The interesting news is that NTP has apparently being getting away with this for **decades** across a very wide span of hardware, operating systems, and compilers, including not just Unixes but under Windows variants as well. This suggests that platforms with padding rules other than self-alignment are either nonexistent or confined to such specialized niches that they’re never either NTP servers or clients.

## 4. Padding

Now we’ll look at a simple example of variable layout in memory. Consider the following series of variable declarations in the top level of a C module:

char \*p;

char c;

int x;

If you didn’t know anything about data alignment, you might assume that these three variables would occupy a continuous span of bytes in memory. That is, on a 32-bit machine 4 bytes of pointer would be immediately followed by 1 byte of char and that immediately followed by 4 bytes of int. And a 64-bit machine would be different only in that the pointer would be 8 bytes.

In fact, the hidden assumption that the allocated order of static variables is their source order is not necessarily valid; the C standards don’t mandate it. I’m going to ignore this detail because (a) that hidden assumption is usually correct anyway, and (b) the actual purpose of talking about padding and packing outside structures is to prepare you for what happens inside them.

Here’s what actually happens (on an x86 or ARM or anything else with self-aligned types). The storage for p starts on a self-aligned 4- or 8-byte boundary depending on the machine word size. This is *pointer alignment* - the strictest possible.

The storage for c follows immediately. But the 4-byte alignment requirement of x forces a gap in the layout; it comes out as though there were a fourth intervening variable, like this:

char \*p; /\* 4 or 8 bytes \*/

char c; /\* 1 byte \*/

char pad[3]; /\* 3 bytes \*/

int x; /\* 4 bytes \*/

The pad[3] character array represents the fact that there are three bytes of waste space in the structure. The old-school term for this was "slop". The value of the padding bits is undefined; in particular it is not guaranteed that they will be zeroed.

Compare what happens if x is a 2-byte short:

char \*p;

char c;

short x;

In that case, the actual layout will be this:

char \*p; /\* 4 or 8 bytes \*/

char c; /\* 1 byte \*/

char pad[1]; /\* 1 byte \*/

short x; /\* 2 bytes \*/

On the other hand, if x is a long on a 64-bit machine

char \*p;

char c;

long x;

we end up with this:

char \*p; /\* 8 bytes \*/

char c; /\* 1 byte

char pad[7]; /\* 7 bytes \*/

long x; /\* 8 bytes \*/

If you have been following carefully, you are probably now wondering about the case where the shorter variable declaration comes first:

char c;

char \*p;

int x;

If the actual memory layout were written like this

char c;

char pad1[M];

char \*p;

char pad2[N];

int x;

## 5. Structure alignment and padding

In general, a struct instance will have the alignment of its widest scalar member. Compilers do this as the easiest way to ensure that all the members are self-aligned for fast access.

Also, in C the address of a struct is the same as the address of its first member - there is no leading padding. Beware: in C++, classes that look like structs may break this rule! (Whether they do or not depends on how base classes and virtual member functions are implemented, and varies by compiler.)

(When you’re in doubt about this sort of thing, ANSI C provides an offsetof() macro which can be used to read out structure member offsets.)

Consider this struct:

struct foo1 {

char \*p;

char c;

long x;

};

Assuming a 64-bit machine, any instance of struct foo1 will have 8-byte alignment. The memory layout of one of these looks unsurprising, like this:

struct foo1 {

char \*p; /\* 8 bytes \*/

char c; /\* 1 byte

char pad[7]; /\* 7 bytes \*/

long x; /\* 8 bytes \*/

};

It’s laid out exactly as though variables of these types has been separately declared. But if we put c first, that’s no longer true.

struct foo2 {

char c; /\* 1 byte \*/

char pad[7]; /\* 7 bytes \*/

char \*p; /\* 8 bytes \*/

long x; /\* 8 bytes \*/

};

If the members were separate variables, c could start at any byte boundary and the size of pad might vary. Because struct foo2 has the pointer alignment of its widest member, that’s no longer possible. Now c has to be pointer-aligned, and following padding of 7 bytes is locked in.

Now let’s talk about trailing padding on structures. To explain this, I need to introduce a basic concept which I’ll call the *stride address* of a structure. It is the first address following the structure data that has the **same alignment as the structure**.

16. Union Inside a structure is possible, how?

# Anonymous Union and Structure in C

In [C11](https://en.wikipedia.org/wiki/C11_(C_standard_revision)) standard of C, anonymous Unions and structures were added.

Anonymous unions/structures are also known as unnamed unions/structures as they don’t have names. Since there is no names, direct objects(or variables) of them are not created and we use them in nested structure or unions.

Definition is just like that of a normal union just without a name or tag. For example,

// Anonymous union example

union

{

char alpha;

int num;

};

// Anonymous structure example

struct

{

char alpha;

int num;

};

Since there is no variable and no name, we can directly access members. This accessibility works only inside the scope where the anonymous union is defined.

Following is a complete working example of anonymous union.

|  |
| --- |
| // C Program to demonstrate working of anonymous union  #include<stdio.h>  struct Scope  {      // Anonymous union      union      {          char alpha;          int num;      };  };    int main()  {      struct Scope x;      x.num = 65;        // Note that members of union are accessed directly      printf("x.alpha = %c, x.num = %d", x.alpha, x.num);        return 0;  } |

Run on IDE

Output:

x.alpha = A, x.num = 65

|  |
| --- |
| // C Program to demonstrate working of anonymous struct  #include<stdio.h>  struct Scope  {      // Anonymous structure      struct      {          char alpha;          int num;      };  };    int main()  {      struct Scope x;      x.num = 65;      x.alpha = 'B';        // Note that members of structure are accessed directly      printf("x.alpha = %c, x.num = %d", x.alpha, x.num);        return 0;  } |

Run on IDE

Output:

x.alpha = B, x.num = 65

# What is practical problem if we use Union inside a Structure and Vice - Verse?

What is practical problem in running application if we use Union inside a Structure and Structure inside a Union?

N/A

Unions basically represent two or more different views of the same data.

A good example would be a data record in a file (organized with a fixed record length). One way of looking at the record, say, would be as a sequence of 16 bytes. Another way would be to look at it as structured data. Something like this:

union

{

  unsigned char raw\_data[16];

  struct

  {

    char name[12];

    short age;

    short height;

  }

}

Code that reads or writes data from/to disk would be using the raw\_data member. Code that actually interprets the data (e.g., presents it in a UI) would be using the

----------------------------------------------------------------------------------

Unions basically represent two or more different views of the same data.

A good example would be a data record in a file (organized with a fixed record length). One way of looking at the record, say, would be as a sequence of 16 bytes. Another way would be to look at it as structured data. Something like this:

union

{

  unsigned char raw\_data[16];

  struct

  {

    char name[12];

    short age;

    short height;

  }

}

Code that reads or writes data from/to disk would be using the raw\_data member. Code that actually interprets the data (e.g., presents it in a UI) would be using the

Qualcomm

It is just a design choice whether to put union inside the structure or a structure inside a union. Similar to the example given by Toth there is a union in Linux data structures also. Which is IP address. It is:

union ipv4addr {

unsigned addr;

char octets[4];

};

Generally we want to refer to the 32-bit value directly, at that time we can use the addr part of the API or if we want to access individual bytes of the IP address then we can use octets. Thus it is the way of giving different meaning to a single API.

17. Segmentation Fault or access violation in c

|  |  |
| --- | --- |
|  | A Segmentation Fault happens when your program tries to access memory that was not allocated to that program. |

The most common reason for a Segmentation Fault (except dereferencing a NULL pointer) is to access an array beyond its bounds.

f.ex:

int arr[5];

for (int i=0; i<=5; i++)

arr[i]=i;

will throw an Segmentation fault because you access the 5th element of arr which does not exist (and thus you try to access the memory behind it which was not allocated to you.

There are multiple places in your program where this can happen.

void rearrangeMainDiagonal(int mat[MAX\_ORDER][MAX\_ORDER], int order)

{

int i, j, k=0, l=0, n=0;

you create fixed sized arrays but you never check your index when you use them. if all my other adjustments are correct it is better to use:

int temp[MAX\_ORDER], odd\_temp[MAX\_ORDER], even\_temp[MAX\_ORDER];

and enforce order to be lower or equal to MAX\_ORDER:

assert(order <= MAX\_ORDER);

based on the function name I suspect that this

for(i=0;i<order;i++)

{

for(j=0;j<order;j++)

{

temp[k] = mat[i][i];

k++;

}

}

which requires temp to be order\*order in size;

should be more like

for(i=0;i<order;i++)

{

temp[i] = mat[i][i];

}

thus placing every element on the main diagonal once in the temp array which now should only be order in size

here you loop over temp until the k'th element, which you did not set in your version of the above loop since you did increment k after the assignment so you should loop until k-1 so use i<kinstead of i<=k;

for(i=0;i<=k;i++)

should become (after the change in the loop above);

for(i=0;i<order;i++)

{

if(temp[i]%2==0)

{

even\_temp[l] = temp[i];

l++;

}

else

{

odd\_temp[n] = temp[i];

n++;

}

}

again the n'th element of odd\_temp is not set, use j<n

for(j=0;j<n;j++)

{

temp[j] = odd\_temp[j];

}

again the l'th element of even\_temp is not set, use i<l

for(i=0;i<l;i++,j++)

{

temp[j] = even\_temp[i];

}

here the same mistake as in the first loop happens. this should become:

for(i=0;i<order;i++)

{

mat[i][i] = temp[i];

}

}

now you can also remove the variable k since it is unused, and if the function does still do what you wanted it to do, it should be able to handle matrices with order up to MAX\_ORDER

# Troubleshooting Segmentation Violations/Faults

A common run-time error for C programs by beginners is a "segmentation violation" or "segmentation fault." When you run your program and the system reports a "segmentation violation," it means your program has attempted to access an area of memory that it is not allowed to access. In other words, it attempted to stomp on memory ground that is beyond the limits that the operating system (e.g., Unix) has allocated for your program.

Any time your program gives a "segmentation violation" or "segmentation fault" error, review this document for tips on correcting the error.

## Common causes of this problem:

* **Improper format control string in printf or scanf statements:**  
  Make sure the format control string has the same number of conversion specifiers (%'s) as the printf or scanf has arguments to be printed or read, respectively, and that the specifiers match the type of variable to be printed or read. This also applies to fprintf and fscanf.
* **Forgetting to use "&" on the arguments to scanf:**  
  Function scanf takes as arguments the format control string and the addresses of variables in which it will place the data that it reads in. The "&" (address of) operator is used to supply the address of a variable. It is common to forget to use "&" with each variable in a scanf call. Omitting the "&" can cause a segmentation violation.
* **Accessing beyond the bounds of an array:**  
  Make sure that you have not violated the bounds of any array you are using; i.e., you have not subscripted the array with a value less than the index of its lowest element or greater than the index of its highest element.
* **Failure to initialize a pointer before accessing it:**  
  A pointer variable must be assigned a valid address (i.e., appear on the left-hand-side of an assignment) before being accessed (i.e., appearing on the right-hand-side of an assignment). Make sure that you have initialized all pointers to point to a valid area of memory. Proper pointer initialization can be done several ways. Examples are listed below.
* **Incorrect use of the "&" (address of) and "\*" (dereferencing) operators:**  
  Make sure you understand how these operators work. Know when they should be applied and when not to apply them. As mentioned above, it is common to forget to use "&" with each variable in a scanf call. Remember, scanf requires the address of the variables it is reading in. Especially, know when "&" and "\*" are absolutely necessary and when it is better to avoid using them.

## Proper pointer initialization:

One common way is to assign the pointer an address to a previously defined variable. For example:

int \* ptr;

int variable;

ptr = &variable;

Or, equivalently,

int variable;

int \*ptr = &variable;

Other common ways include assigning the pointer the address of memory allocated with matrix, vector, calloc, or malloc or other equivalent allocation functions. Remember, a pointer must be initialized to a value (i.e., assigned a value by appearing on the left-hand-side of an assignment statement) BEFORE you attempt to access it!

## Minimizing the use of pointer variables.

Also, many times a function requires that an address (corresponding to a parameter of pointer type) be sent to it as an argument (as is true of many of the Numerical Recipes in C functions). The standard function scanf is an example of such a function.   
In these cases, it is usually best to simply declare a variable of the correct type before calling the function and just sending the address of the variable to the function. In fact, that is what is intended in the vast majority of these cases. And, that's how you usually scanf:

double x\_initial; /\* initial guess \*/

scanf("%lf",&x\_initial); /\* Read the initial guess. \*/

For example, see how 'idum' is used below:

long idum = -1; /\* initialize idum to be a negative integer \*/

/\* generate a random number from the normal distn.\*/

x = normal(&idum,average,stddev);

The function normal expects an address to a variable of type long. That's what we send it without explicitly using a pointer variable in the calling routine.

## Troubleshooting the problem:

Check EVERY place in your program that uses pointers, subscripts an array, or uses the address operator (&) and the dereferencing operator (\*). Each is a candidate for being the cause of a segmentation violation. Make sure that you understand the use of pointers and the related operators.

**If the program uses many pointers and has many occurrences of & and \*, then add some printf statements to pinpoint the place at which the program causes the error and investigate the pointers and variables involved in that statement.**

# ccess Violation

You may see messages containing "access violation" words when segmentation faults occur.

A segmentation fault (segfault in abbreviated form) is a software error occurring when a program tries to access memory addresses unavailable for writing or when a program tries to modify memory using an illegal method.

Segmentation is one of the approaches to memory management and protection in an operating system. In most systems it has been replaced by paged memory, but documentations traditionally use the term "Segmentation fault".

In UNIX-like operating systems, a process accessing invalid memory addresses receives a SIGSEGV signal. In Microsoft Windows, a process accessing invalid memory addresses raises an exception STATUS\_ACCESS\_VIOLATION and usually launches the Dr. Watson program which shows the user a window prompting to send the error report to Microsoft.

Memory access violation is most often caused by such errors in programs as array overruns or usage of a null pointer.

Let's examine a defect in a C++ program that can cause this type of errors. This error was found by our analyzer [PVS-Studio](https://www.viva64.com/en/pvs-studio/) in the [Chromium](http://www.chromium.org/) project.

bool ChromeFrameNPAPI::Invoke(...)

{

ChromeFrameNPAPI\* plugin\_instance =

ChromeFrameInstanceFromNPObject(header);

if (!plugin\_instance &&

(plugin\_instance->automation\_client\_.get()))

return false;

...

}

This code should check the value of the 'plugin\_instance' pointer and call the function if the pointer is not equal to zero. The error here is that the [priority of the operator](https://www.viva64.com/en/t/0064/) '!' is higher than that of the '&&' operator. As a result, the code behaves in an unexpected way. Arranging parentheses clarifies the point:

if ( (!plugin\_instance) &&

(plugin\_instance->automation\_client\_.get()))

return false;

It turns out that we will use a null pointer. Handling a null pointer will cause a segmentation fault and an exception will be thrown.

**17. What is a page fault in an operating system?**

To understand what a page fault is, you need to understand virtual memory.

Virtual memory is a way of mapping a “fake” memory address to a “real” memory address. The “real” address (the “physical address”) is what your hardware actually uses to read and write to RAM, while the “fake” address (the “virtual address”) is what your program thinksthose addresses are.

When your program reads/writes a memory location, it will use a virtual address to refer to that location in memory. Your system will then translate that virtual address into a physical address, and the data will be written to (or read from) that physical address. But your program has no idea that this is happening— the virtual to physical address translation is all kind of “under the hood”. The point is, your program doesn’t need to know that this is happening. The system will just maintain a happy illusion for the program to live in.

There are a number of reasons why you’d want to have this system; nearly all memory systems today use virtual memory, except in certain specialized situations. One major advantage of virtual memory is that you can use it to “pretend” that your machine has more RAM than it actually has installed— this usually happens through a system called “paging” (sometimes called “swapping”).

The idea is that the physical RAM only holds a certain number of “pages” of memory. Typically a “page” is about 4096 bytes, although it varies from system to system. In order to “pretend” that there is more RAM than physically possible, some of the pages are actually in the RAM, and some of them are stored on the hard drive.

The virtual memory system uses something called a “page table” to map virtual addresses to physical addresses. Since our machine could possibly have less RAM than our program thinks it has, it’s possible to have more virtual addresses than physical addresses. That means not all virtual addresses in a page table will have a valid corresponding physical address (i.e. not all virtual addresses will have a valid entry in the page table). If a virtual address has no valid entry in the page table, then any attempt by your program to access that virtual address will cause a page fault to occur— if you’re familiar with software exceptions (like in C++, Java, Python, etc.), a page fault is very much like an exception, except in hardware, rather than software. The page fault happens because the requested virtual address actually corresponds to a page that is currently sitting on disk, rather than in RAM (and therefore the virtual address cannot possibly be translated into a physical address).

So what happens when we have a page fault? Well, when that happens, your OS invokes something called a **page fault handler**. As you might imagine, it’s a piece of code that handles page faults. Usually the page fault handler will do the following:

Figure out which page the virtual address is supposed to map to. Figure out where that page is located on the hard drive.

Choose an existing page in physical RAM that we (probably) aren’t currently using. Write that page back to the hard drive, and evict it from RAM (i.e. kick it out) to make room for the new page.

Load the new page into RAM, from the hard drive.

Update the page table, so that the virtual address that caused the page fault now has a valid entry. Likewise, clear the virtual address entry for the page that we just evicted, so that virtual addresses that correspond to the evicted page are no longer valid.

Now that the correct page is loaded in RAM, and the page table is up to date, return control to the program that was running before all these shenanigans occurred, and retry the memory access instruction that initially caused the page fault. If all goes well, this second time around the faulting instruction will work correctly, now that the correct page has been loaded into physical RAM.

Of course, this is a very, very simplistic oversimplification of the many design considerations that go into virtual memory, but it at least gives the basic idea of what a page fault is.

# Page Fault

A page fault occurs when a [program](https://techterms.com/definition/program) attempts to access a block of [memory](https://techterms.com/definition/memory) that is not stored in the physical memory, or [RAM](https://techterms.com/definition/ram). The fault notifies the [operating system](https://techterms.com/definition/operating_system) that it must locate the data in [virtual memory](https://techterms.com/definition/virtualmemory), then transfer it from the storage device, such as an [HDD](https://techterms.com/definition/hdd) or [SSD](https://techterms.com/definition/ssd), to the system RAM.

Though the term "page fault" sounds like an error, page faults are common and are part of the normal way computers handle virtual memory. In programming terms, a page fault generates an [exception](https://techterms.com/definition/exception), which notifies the operating system that it must retrieve the memory blocks or "pages" from virtual memory in order for the program to continue. Once the [data](https://techterms.com/definition/data) is moved into physical memory, the program continues as normal. This process takes place in the background and usually goes unnoticed by the user.

Most page faults are handled without any problems. However, an invalid page fault may cause a program to hang or [crash](https://techterms.com/definition/crash). This type of page fault may occur when a program tries to access a memory address that does not exist. Some programs can handle these types of errors by finding a new memory address or relocating the data. However, if the program cannot handle the invalid page fault, it will get passed to the operating system, which may terminate the [process](https://techterms.com/definition/process). This can cause the program to unexpectedly quit.

While page faults are common when working with virtual memory, each page fault requires transferring data from [secondary memory](https://techterms.com/definition/secondary_memory) to [primary memory](https://techterms.com/definition/primary_memory). This process may only take a few milliseconds, but that can still be several thousand times slower than accessing data directly from memory. Therefore, installing more system memory can increase your computer's performance, since it will need to access virtual memory less often.

18. Re-etrant function in c

Put at its simplest (or as simple as I cn make it, a re-entrant function is one that can be re-entered before it's actually finished executing.

One typical situation is with interrupts, where you may be in the middle of a function when an interrupt occurs, then the interrupt service routine calls that function as part of its workload.

Another is with recursion, where a function calls itself (either directly or indirectly).

Re-entrant functions have to follow certain guidelines (such as no use of static variables in C) lest different instances of them trample on each other.

Function is called reentrant if it can be interrupted in the middle of its execution and then safely called again ("re-entered") before its previous invocations complete execution

What makes one function not being re-entrant? Check the article further, but roughly:

* Do not use static or global variables in your function since those may be changed by time your function resumes
* Function must not modify its own code (e.g. some low level graphic routines may have "habit" to generate itself)
* Do not call any function that does not comply with the two rules above
* When to use re-entrant function? Here are some examples:
  + Functions executed in interrupt context must be re-entrant.
  + Functions that will be called from multiple threads/tasks must be re-entrant.

# Reentrant Function

A function is said to be reentrant if there is a provision to interrupt the function in the course of execution, service the interrupt service routine and then resume the earlier going on function, without hampering its earlier course of action. Reentrant functions are used in applications like hardware interrupt handling, recursion, etc.

   
The function has to satisfy certain conditions to be called as reentrant:

1. It may not use global and static data. Though there are no restrictions, but it is generally not advised. because the interrupt may change certain global values and resuming the course of action of the reentrant function with the new data may give undesired results

.  
  
   
2. It should not modify is own code. This is important because the course of action of the function should remain the same throughout the code. But, this may be allowed in case the interrupt routine uses a local copy of the reentrant function every time it uses different values or before and after the interrupt.

3. Should not call another non-reentrant function.

**Thread safety and Reentrant functions**

Reentrancy is distinct from, but closely related to, thread-safety. A function can be thread-safe and still not reentrant. For example, a function could be wrapped all around with a mutex (which avoids problems in multithreading environments), but if that function is used in an interrupt service routine, it could starve waiting for the first execution to release the mutex. The key for avoiding confusion is that reentrant refers to only one thread executing. It is a concept from the time when no multitasking operating systems existed. (Source

:<https://en.wikipedia.org/wiki/Reentrancy_(computing)>)

Example of Non-Reentrant Functions:

|  |
| --- |
| // A non-reentrant example [The function depends on  // global variable i]    int i;    // Both fun1() and fun2() are not reentrant    // fun1() is NOT reentrant because it uses global variable i  int fun1()  {      return i \* 5;  }    // fun2() is NOT reentrant because it calls a non-reentrant  // function  int fun2()  {     return fun1() \* 5;  } |

**When is a function reentrant?**  
  
A function is **reentrant** if it can be invoked while already in the process of executing. That is, a function is reentrant if it can be interrupted in the middle of execution (for example, by a signal or interrupt) and invoked again before the interrupted execution completes.  
  
For example, the following function is not reentrant, because the observed value of the summation depends on when and where the function is interrupted (or, in the case of multithreading, how two or more threads race into the function):

1. static int sum = 0;
3. int increment(int i) {
4. sum += i;
5. return sum;
6. }

We can make this function reentrant by making the sum not a global variable and instead requiring the caller to maintain it:

1. int increment(int sum, int i) {

return sum + i;

}

ctime(), gmtime(), and strtok() are examples of functions in Standard C that are (or at least often are) not reentrant. Later versions of the standard have created reentrant versions, for example strtok\_r(), with the primary difference that the caller passes in storage instead of the function maintaining it globally, as above.  
  
**How does that relate to it being thread-safe?**  
  
The term reentrancy predates threads and is often used only in a single threaded context, for example when discussing if a function is signal or interrupt-safe. Reentrancy and thread-safety are not the same and should not be confused.  
  
Although a reentrant function is more likely to be thread-safe all else equal, a reentrant function **need** not be thread-safe, and a thread-safe function **need not** be reentrant. For example, reentrant functions that operate on the same data (same inputs) may not be thread-safe. Similarly, a thread-safe function that achieves safety through a mutex is potentially not reentrant, as the interruption could deadlock on the already-acquired mutex.  
  
To summarize, we might define reentrancy and thread safety in similar terms, but with slightly different semantics:  
  
A reentrant function can be invoked, interrupted, and re-invoked. Such a function can be invoked simultaneously by multiple threads if and only if each invocation references or provides unique data and inputs.  
  
A thread-safe function can be invoked simultaneously by multiple threads, even if each invocation references or provides the same data or input, as all access is serialized.  
  
**Writing reentrant code.**  
  
Writing code that is reentrant is not hard. The styles and patterns adapted by programmers since the mid-1990s are largely reentrant. Indeed, most programmers today balk at solutions to problems such as the approach adopted by strtok(), even if not concerned with reentrancy or thread safety.  
  
Basic guidelines:

* Do not access mutable global or function-static variables.
* Do not self-modify code.
* Do not invoke another function that is itself non-reentrant.

**Writing reentrant and threadsafe code**

In single-threaded processes, only one flow of control exists. The code executed by these processes thus need not be reentrant or threadsafe. In multithreaded programs, the same functions and the same resources may be accessed concurrently by several flows of control.

To protect resource integrity, code written for multithreaded programs must be reentrant and threadsafe.

Reentrance and thread safety are both related to the way that functions handle resources. Reentrance and thread safety are separate concepts: a function can be either reentrant, threadsafe, both, or neither.

This section provides information about writing reentrant and threadsafe programs. It does not cover the topic of writing thread-efficient programs. Thread-efficient programs are efficiently parallelized programs. You must consider thread effiency during the design of the program. Existing single-threaded programs can be made thread-efficient, but this requires that they be completely redesigned and rewritten.

**Reentrance**

A reentrant function does not hold static data over successive calls, nor does it return a pointer to static data. All data is provided by the caller of the function. A reentrant function must not call non-reentrant functions.

A non-reentrant function can often, but not always, be identified by its external interface and its usage. For example, the [**strtok**](https://www.ibm.com/support/knowledgecenter/ssw_aix_61/com.ibm.aix.basetrf2/strlen.htm?view=kc) subroutine is not reentrant, because it holds the string to be broken into tokens. The [**ctime**](https://www.ibm.com/support/knowledgecenter/ssw_aix_61/com.ibm.aix.basetrf1/ctime.htm?view=kc) subroutine is also not reentrant; it returns a pointer to static data that is overwritten by each call.

**Thread safety**

A threadsafe function protects shared resources from concurrent access by locks. Thread safety concerns only the implementation of a function and does not affect its external interface.

In C language, local variables are dynamically allocated on the stack. Therefore, any function that does not use static data or other shared resources is trivially threadsafe, as in the following example:

/\* threadsafe function \*/

int diff(int x, int y)

{

int delta;

delta = y - x;

if (delta < 0)

delta = -delta;

return delta;

}

Copy to clipboard

The use of global data is thread-unsafe. Global data should be maintained per thread or encapsulated, so that its access can be serialized. A thread may read an error code corresponding to an error caused by another thread. In AIX®, each thread has its own errno value.

**Making a function reentrant**

In most cases, non-reentrant functions must be replaced by functions with a modified interface to be reentrant. Non-reentrant functions cannot be used by multiple threads. Furthermore, it may be impossible to make a non-reentrant function threadsafe.

**Returning data**

Many non-reentrant functions return a pointer to static data. This can be avoided in the following ways:

* Returning dynamically allocated data. In this case, it will be the caller's responsibility to free the storage. The benefit is that the interface does not need to be modified. However, backward compatibility is not ensured; existing single-threaded programs using the modified functions without changes would not free the storage, leading to memory leaks.
* Using caller-provided storage. This method is recommended, although the interface must be modified.

# 20 [what is the difference between re-entrant function and recursive function in C?](https://stackoverflow.com/questions/261311/what-is-the-difference-between-re-entrant-function-and-recursive-function-in-c)

|  |  |
| --- | --- |
|  | A function is re-entrant if it supports having multiple threads of execution "going through" it at the same time. This might be due to actual multi-threading, and I use this case below, or due to other things as pointed out by other posters. Multi-threading was the first that came to mind, and is perhaps also the easiest to understand, so I focused on that case.  This means that the function cannot use static "global" data, since that data would then be accessed by two (or more) threads in parallel, often breaking horribly. A re-entrant function often has an explicit argument to hold any call-specific state, rather than storing it statically.  strtok() is a classic case of a function in the C standard library that is well-known not to be re-entrant. |

The term "re-entrant" means that it is safe to "**re-enter**" the function while it is already executed, typically in a concurrent environment.

In other words, when two tasks can execute the function at the same time without interfering with each other, then the function is re-entrant. A function is not re-entrant when the execution by one task has an impact on the influence of another task. This typically is the case when a global state or data is used. A function that uses only local variables and arguments is typically re-entrant.

* A reentrant function can be called simultaneously by multiple threads provided that each invocation of the function references unique data.
* A thread-safe function can be called simultaneously by multiple threads when each invocation references shared data. All access to the shared data is serialized.

Shamelessly stolen from the Qt manual. But it's a short and concise definition. Basicially, a non-reentrant function is also not recursion-safe.

Now, what is a recursive function? It's a kind of definition of a function. Recursive function are defined in terms of themself. They reduce input, call theirself, until a basic case can be figured out without the need of calling theirself again.

So we have two things.

* recursive functions are a kind of definition.
* reentrant functions are functions that guarantee multiple threads can call them, provided each time unique data is accessed.

Now, the multiple-threads vehicle above serves only the purpose of having multiple activations of the function at the same time. But if you have a recursive function, you *also* have multiple activations of that functions at the same time. Most recursive functions therefor must be re-entrant too.

1. **DIFFERENCE BETWEEN C AND EMBEDDED C**

Though **C and embedded C** appear different and are used in different contexts, they have more similarities than the differences. Most of the constructs are same; the difference lies in their applications.

C is used for desktop computers, while **embedded C** is for microcontroller based applications. Accordingly, C has the luxury to use resources of a desktop PC like memory, OS, etc. While programming on desktop systems, we need not bother about memory. However, embedded C has to use with the limited resources (RAM, ROM, I/Os) on an embedded processor. Thus, program code must fit into the available program memory. If code exceeds the limit, the system is likely to crash.

Compilers for C (ANSI C) typically generate OS dependant executables. **Embedded C** requires compilers to create files to be downloaded to the microcontrollers/microprocessors where it needs to run. Embedded compilers give access to all resources which is not provided in compilers for desktop computer applications.

Embedded systems often have the real-time constraints, which is usually not there with desktop computer applications.

Embedded systems often do not have a console, which is available in case of desktop applications.

So, what basically is different while programming with **embedded C** is the mindset; for embedded applications, we need to optimally use the resources, make the program code efficient, and satisfy real time constraints, if any. All this is done using the basic constructs, syntaxes, and function libraries of ‘C’.

* C is generally used for desktop computers, while embedded C is for microcontroller based applications.
* C can use the resources of a desktop PC like memory, OS, etc. While, embedded C has to use with the limited resources, such as RAM, ROM, I/Os on an embedded processor.
* Embedded C includes extra features over C, such as fixed point types, multiple memory areas, and I/O register mapping.
* Compilers for C (ANSI C) typically generate OS dependent executables; commonly, a file with .exe extension. Embedded C requires compilers to create files that can be downloaded to the microcontrollers/microprocessors where it needs to run. It need not be a .hex file always.