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).***

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

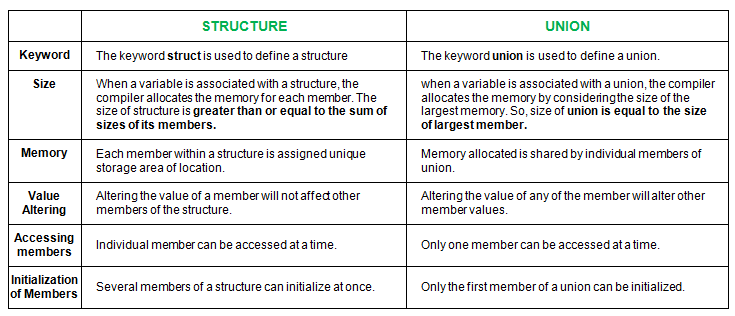
MACRO1 Defined

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).***

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(object).***

***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);

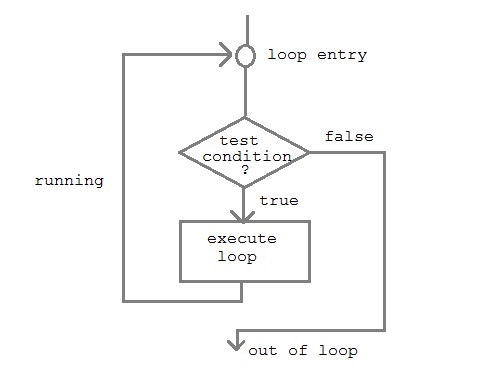
}

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 **for**loop 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) {

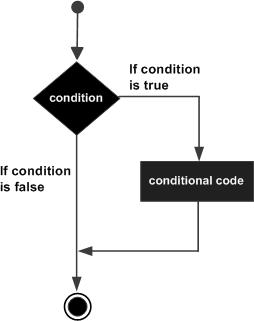
/\* statement(s) will execute if the boolean expression is true \*/

}

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 {

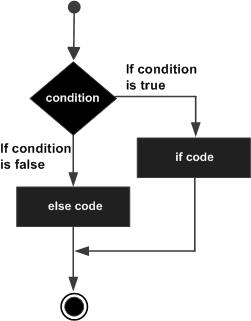
/\* statement(s) will execute if the boolean expression is false \*/

}

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