

Interview Preparation in C Language

Table Of Content:

0. Important Links
1. Data Types
2. Void main() vs Int main()
3. Behind the scenes and compiling
 - a. Linker and Loader
 - b. Macros and Preprocessors
 - c. 32 Bit vs 64 bit gcc
 - d. Platform dependency and independency.
 - e. Internal and External linkage
 - f. How compilation of prog takes place.
4. Error and Exception
5. Error handling
6. Scope
7. Functions
 - a. Static functions
 - b. Overloading
 - c. Call back functions
8. Pointers
 - a. Types
 - b. Double Pointer
 - c. Reference and dereference
 - d. Pointer to Function
 - e. Void / Null / Dangling / Wild
 - f. Near / Far / Huge
 - g. Restrict keyword
9. Storage class
 - a. Static keyword
 - b. Volatile
 - c. Register
10. Enum, struct and Union
11. Bit Fields
12. Malloc and Calloc
13. Deallocation of memory
 - a. With free
 - b. Without free
14. Operators
15. Precedence Table
16. File Handling

- 17. Preprocessor Directive
 - 18. _Generic Keyword
 - 19. Multithreading
 - 20. Assertion
 - 21. Control Statement
 - a. If else
 - b. switch
 - c. Goto
 - 22. Hygienic Macros
-

0. Important Links:

[Geeks for geeks - C](#)

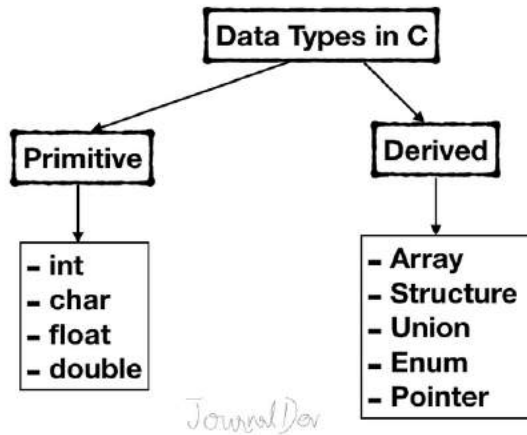
[2braces I/O Questions](#)

[Includehelp I/O Questions](#)

1. Data Types :

There are the following data types in C language.

Types	Data Types
Basic Data Type	int, char, float, double
Derived Data Type	array, pointer, structure, union
Enumeration Data Type	enum
Void Data Type	void



C Basic Data Types	32-bit CPU		64-bit CPU	
	Size (bytes)	Range	Size (bytes)	Range
char	1	-128 to 127	1	-128 to 127
short	2	-32,768 to 32,767	2	-32,768 to 32,767
int	4	-2,147,483,648 to 2,147,483,647	4	-2,147,483,648 to 2,147,483,647
long	4	-2,147,483,648 to 2,147,483,647	8	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
long long	8	9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	8	9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
float	4	3.4E +/- 38	4	3.4E +/- 38
double	8	1.7E +/- 308	8	1.7E +/- 308

A keyword is a **reserved word**. You cannot use it as a variable name, constant name, etc. There are only 32 reserved words (keywords) in the C language.

A list of 32 keywords in the c language is given below:

auto	break	case	char	const	continue	default	do
double	else	enum	extern	float	for	goto	if
int	long	register	return	short	signed	sizeof	static
struct	switch	typedef	union	unsigned	void	volatile	while

Differences between Keyword and Identifier

Keyword	Identifier
Keyword is a pre-defined word.	The identifier is a user-defined word
It must be written in a lowercase letter.	It can be written in both lowercase and uppercase letters.
Its meaning is pre-defined in the c compiler.	Its meaning is not defined in the c compiler.
It is a combination of alphabetical characters.	It is a combination of alphanumeric characters.
It does not contain the underscore character.	It can contain the underscore character.

2. Void main() vs Int main():

A conforming implementation may provide more versions of `main()`, but they must all have return type `int`. The `int` returned by `main()` is a way for a program to return a value to “the system” that invokes it. On systems that doesn’t provide such a facility the return value is ignored, but that doesn’t make “`void main()`” legal C++ or legal C. ***Even if your compiler accepts “void main()” avoid it, or risk being considered ignorant by C and C++ programmers.***

In C++, `main()` need not contain an explicit return statement. In that case, the value returned is 0, meaning successful execution.

3. Behind the scenes and compiling:

What goes inside the compilation process?

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

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

Pre-processing

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

- Removal of Comments
- Expansion of Macros
- Expansion of the included files.
- Conditional compilation

Compiling

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

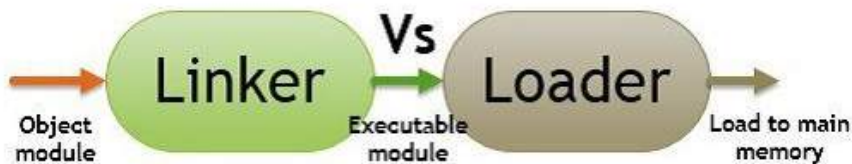
Assembly

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

Linking

This is the final phase in which all the linking of function calls with their definitions are done. Linker knows where all these functions are implemented. Linker does some extra work also, it adds some extra code to our program which is required when the program starts and ends. For example, there is a code which is required for setting up the environment like passing command line arguments. This task can be easily verified by using **`$size filename.o`** and **`$size filename`**. Through these commands, we know that how output file increases from an object file to an executable file. This is because of the extra code that linker adds with our program.

a. Linker and Loader:



BASIS FOR COMPARISON	LINKER	LOADER
Basic	It generates the executable module of a source program.	It loads the executable module to the main memory.
Input	It takes as input, the object code generated by an assembler.	It takes executable module generated by a linker.
Function	It combines all the object modules of a source code to generate an executable module.	It allocates the addresses to an executable module in main memory for execution.
Type/Approach	Linkage Editor, Dynamic linker.	Absolute loading, Relocatable loading and Dynamic Run-time loading.

Key Differences Between Linker and Loader

1. The key difference between linker and loader is that the linker generates the **executable** file of a program whereas, the loader loads the executable file obtained from the linker into **main memory for execution**.
2. The linker intakes the **object module** of a program generated by the assembler. However, the loader intakes the **executable module** generated by the linker.
3. The linker combines all object module of a program to generate **executable modules** it also links the **library function** in the object module to **built-in libraries** of the high-level programming language. On the other hands, loader **allocates space to an executable module** in main memory.
4. The linker can be classified as **linkage editor**, and **dynamic linker** whereas loader can be classified as **absolute loader**, **relocatable loader** and **dynamic run-time loader**.

b. Macros and Preprocessors:

Sr.No	Directives & Descriptions
1	#define Substitutes a preprocessor macro.
2	#include Inserts a particular header from another file.
3	#undef Undefines a preprocessor macro.
4	#ifdef Returns true if this macro is defined.
5	#ifndef Returns true if this macro is not defined.
6	#if Tests if a compile time condition is true.
7	#else The alternative for #if.
8	#elif #else and #if in one statement.
9	#endif Ends preprocessor conditional.
10	#error Prints error message on stderr.
11	#pragma Issues special commands to the compiler, using a standardized method.

c. 32 Bit vs 64 bit machine:

[Computerhope](#)

[Gfg](#)

d. Platform dependency and independency:

For the case of C and C++

Lets take example of Linux and Windows to explain this.

We say that C is platform dependent because

1. if you compile and build a C program code in windows, copy that .exe file to a linux machine, that .exe file will not run there.
2. In the same way if you compile the same program code on linux, you'll get a .out file which will not run on windows if directly copied there.

e. Internal and External linkage:

4. Error and Exception:

Difference in Java :

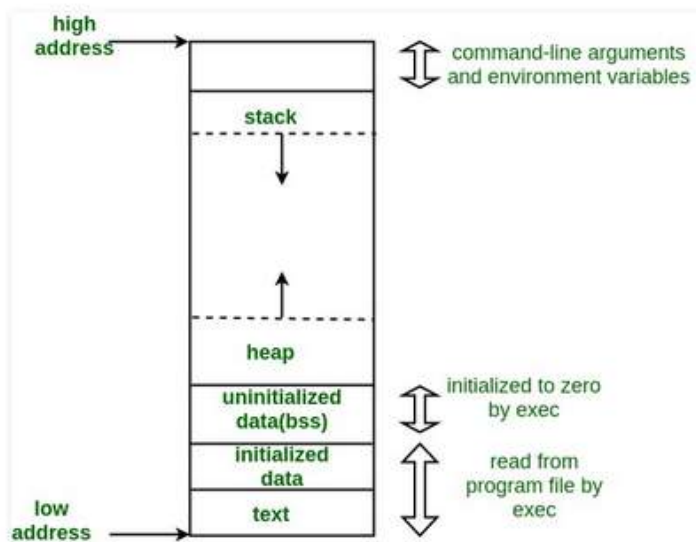
Errors	Exceptions
Recovering from Error is not possible.	We can recover from exceptions by either using try-catch block or throwing exceptions back to caller.
All errors in java are unchecked type.	Exceptions include both checked as well as unchecked type.
Errors are mostly caused by the environment in which program is running.	Program itself is responsible for causing exceptions.
Errors occur at runtime and not known to the compiler.	All exceptions occurs at runtime but checked exceptions are known to compiler while unchecked are not.
They are defined in java.lang.Error package.	They are defined in java.lang.Exception package
Examples : java.lang.StackOverflowError, java.lang.OutOfMemoryError	Examples : Checked Exceptions : SQLException, IOException Unchecked Exceptions : ArrayIndexOutOfBoundsException, NullPointerException, ArithmeticException.

5. Error handling:

6. Scope:

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

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



KEY DIFFERENCE

- Stack is a linear data structure whereas Heap is a hierarchical data structure.
- Stack memory will never become fragmented whereas Heap memory can become fragmented as blocks of memory are first allocated and then freed.
- Stack accesses local variables only while Heap allows you to access variables globally.
- Stack variables can't be resized whereas Heap variables can be resized.
- Stack memory is allocated in a contiguous block whereas Heap memory is allocated in any random order.
- Stack doesn't require to de-allocate variables whereas in Heap de-allocation is needed.
- Stack allocation and deallocation are done by compiler instructions whereas Heap allocation and deallocation is done by the programmer.

When to use the Heap or stack?

You should use heap when you require to allocate a large block of memory. For example, you want to create a large size array or big structure to keep that variable around a long time then you should allocate it on the heap.

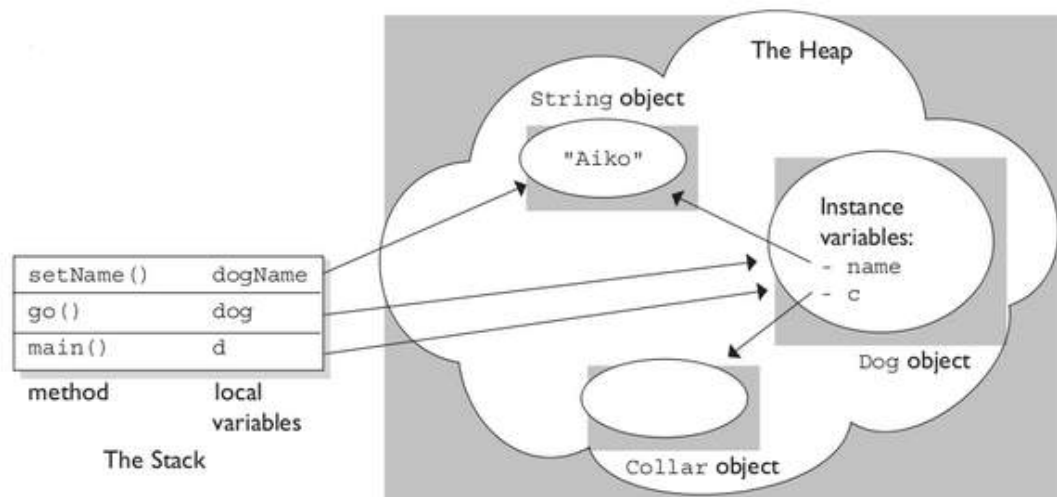
However, If you are working with relatively small variables that are only required until the function using them is alive. Then you need to use the stack, which is faster and easier.

The stack is the memory set aside as scratch space for a thread of execution. When a function is called, a block is reserved on the top of the stack for local variables and some bookkeeping data. When that function returns, the block becomes unused and can be used the next time a function is called. The stack is always reserved in a LIFO (last in first out) order; the most recently reserved block is always the next block to be freed. This makes it really simple to keep track of the stack; freeing a block from the stack is nothing more than adjusting one pointer.

The heap is memory set aside for dynamic allocation. Unlike the stack, there's no enforced pattern to the allocation and deallocation of blocks from the heap; you can allocate a block at any time and free it at any time. This makes it much more complex to keep track of which parts of the heap are allocated or freed at any given time; there are many custom heap allocators available to tune heap performance for different usage patterns.

Each thread gets a stack, while there's typically only one heap for the application (although it isn't uncommon to have multiple heaps for different types of allocation).

A clear demonstration:



Comparison Chart

Parameter	STACK	HEAP
Basic	Memory is allocated in a contiguous block.	Memory is allocated in any random order.
Allocation and De-allocation	Automatic by compiler instructions.	Manual by the programmer.
Cost	Less	More
Implementation	Easy	Hard
Access time	Faster	Slower
Main Issue	Shortage of memory	Memory fragmentation
Locality of reference	Excellent	Adequate
Safety	Thread safe, data stored can only be accessed by owner	Not Thread safe, data stored visible to all threads
Flexibility	Fixed-size	Resizing is possible
Data type structure	Linear	Hierarchical

7. Static keyword:

[Quiz on static](#)

- **Static variables in a Function:** When a variable is declared as static, space for **it gets allocated for the lifetime of the program**. Even if the function is called multiple times, space for the static variable is allocated only once and the value of variable in the previous call gets carried through the next function call. This is useful for implementing [coroutines in C/C++](#) or any other application where previous state of function needs to be stored.

[Coroutines ?](#)

- **Static variables in a class:** As the variables declared as static are initialized only once as they are allocated space in separate static storage so, the static variables **in a class are shared by the objects**. There can not be multiple copies of same static variables for different objects. Also because of this reason static variables can not be initialized using constructors.

Static variables have a property of preserving their value even after they are out of their scope! Hence, static variables preserve their previous value in their previous scope and are not initialized again in the new scope.

Syntax:

```
static data_type var_name = var_value;
```

Following are some interesting facts about static variables in C.

- 1)** A static int variable remains in memory while the program is running. A normal or auto variable is destroyed when a function call where the variable was declared is over.
- 2)** Static variables are allocated memory in data segment, not stack segment. See [memory layout of C programs](#) for details.
- 3)** Static variables (like global variables) are initialized as 0 if not initialized explicitly. For example in the below program, value of x is printed as 0, while value of y is something garbage. See [this](#) for more details.

4) In C, static variables can only be initialized using constant literals. For example, following program fails in compilation. See [this](#) for more details.

```
#include<stdio.h>
int initializer(void)
{
    return 50;
}

int main()
{
    static int i = initializer();
    printf(" value of i = %d", i);
    getchar();
    return 0;
}
```

Output

```
In function 'main':
9:5: error: initializer element is not constant
    static int i = initializer();
    ^
```

5) Static global variables and functions are also possible in C/C++. The purpose of these is to limit scope of a variable or function to a file. Please refer [Static functions in C](#) for more details.

6) Static variables should not be declared inside structure. The reason is C compiler requires the entire structure elements to be placed together (i.e.) memory allocation for structure members should be contiguous. It is possible to declare structure inside the function (stack segment) or allocate memory dynamically(heap segment) or it can be even global (BSS or data segment). Whatever might be the case, all structure members should reside in the same memory segment because the value for the structure element is fetched by counting the offset of the element from the beginning address of the structure. Separating out one member alone to data segment defeats the purpose of static variable and it is possible to have an entire structure as static.

[Can static func be virtual in C++](#)

Internal Static Variables: Internal Static variables are defined as those having static variables which are declared inside a function and extends up to the end of the particular function.

Syntax:

```
main( )
{
    static datatype variable;
    // other statements
}
```

External Static Variables: External Static variables are those which are declared outside a function and set globally for the entire file/program.

```
#include <stdio.h>

int add(int, int);

static int a = 5;

int main()
{
    int c;
    printf("%d", add(a, c));
}

int add(int c, int b)
{
    b = 5;
    c = a + b;
    return c;
}
```

Output:

10

Parameter	Internal Static Variables	External Static Variables
Keyword	"static"	"static"
Linkage	Internal static variable has no linkage.	External static variables has internal linkage.
Declaration	Internal static variables are declared within the main function	External static variables are declared above the main function.
Comparison	Internal static variables are similar to auto(local) variables.	External static variables are similar to global(external) variables.
Visibility	Internal static variables are active(visibility) in the particular function.	External Static variables are active(visibility) throughout the entire program.
Lifetime	Internal static variables are alive(lifetime) until the end of the function.	External static variables are alive(lifetime) in the entire program.
Scope	Internal static variables has persistent storage with block scope(works only within a particular block of function)	External static variables has permanent storage with file scope(works throughout the program).

8. Functions:

Parameter Passing to functions

The parameters passed to function are called **actual parameters**. For example, in the above program 10 and 20 are actual parameters.

The parameters received by function are called **formal parameters**. For example, in the above program x and y are formal parameters.


There are two most popular ways to pass parameters.

Pass by Value: In this parameter passing method, values of actual parameters are copied to function's formal parameters and the two types of parameters are stored in different memory locations. So any changes made inside functions are not reflected in actual parameters of caller.

Pass by Reference Both actual and formal parameters refer to same locations, so any changes made inside the function are actually reflected in actual parameters of caller.

a. Static functions:

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



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

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

There is a big difference between static functions in C and static member functions in C++. In C, a static function is not visible outside of its translation unit, which is the object file it is compiled into. In other words, making a function static limits its scope. You can think of a static function as being "private" to its *.c file (although that is not strictly correct).

In C++, "static" can also apply to member functions and data members of classes. A static data member is also called a "class variable", while a non-static data member is an "instance variable". This is Smalltalk terminology. This means that there is only one copy of a static data member shared by all objects of a class, while each object has its own copy of a non-static data member. So a static data member is essentially a global variable, that is a member of a class.

Non-static member functions can access all data members of the class: static and non-static. Static member functions can only operate on the static data members.

One way to think about this is that in C++ static data members and static member functions do not belong to any object, but to the entire class.

b. Overloading:

Does C support function overloading like C++?

If you consider the `printf()` function in C, that may lead you to think that C supports function overloading. Because, in C you can have `printf("%d", aDecimal)` and `printf("%f", aFloat)`. This looks a lot like function overloading, because we are using the same function name, but the function is accepting different parameter types – which is one of the ways we can overload functions.

A Variable Argument List is not Function Overloading

Actually, this is *not* a case of function overloading – the `printf` function is just using a feature of C known as variable argument lists. This should not be confused with function overloading. So, to answer the question, Standard C does *not* support function overloading.

As an interesting side note, C++ doesn't really have function overloading. What it does have is a means of faking it: the C++ compiler actually 'mangles' (or changes) function names according to the function's parameters. So, functions that share the same name but have different numbers or types of parameters can be differentiated when invoked. Also, since the 'mangling' of function names is not standardized, it's usually difficult to link object files compiled by different C++ compilers.

</> source code

```
1  #include <stdio.h>
2  int add(int a, int b, int c) {
3      return a + b + c;
4  }
5  int add(int a, int b) {
6      return a+b;
7  }
8  int main(void) {
9      printf("%d", add(1, 2, 3));
10     printf("%d", add(1, 2));
11     return 0;
12 }
13
```

Compilation error #stdin compilation error #stdout 0s 5520KB

prog.c:5:5: error: conflicting types for 'add'

```
int add(int a, int b) {
```

^~~

prog.c:2:5: note: previous definition of 'add' was here

```
int add(int a, int b, int c) {
```

^~~

prog.c: In function 'main':

c. Call back functions:

A callback is any executable code that is passed as an argument to other code, which is expected to call back (execute) the argument at a given time [Source : [Wiki](#)]. In simple language, If a reference of a function is passed to another function as an argument to call it, then it will be called as a Callback function.

In C, a callback function is a function that is called through a [function pointer](#).

```
#include<stdio.h>

void A()
{
    printf("I am function A\n");
}

// callback function
void B(void (*ptr)())
{
    (*ptr) (); // callback to A
}

int main()
{
    void (*ptr)() = &A;

    // calling function B and passing
    // address of the function A as argument
    B(ptr);

    return 0;
}
```

```
am function A
```

9. _Generic Keyword:

A major drawback of [Macro in C/C++](#) is that the arguments are strongly typed checked i.e. a macro can operate on different types of variables (like char, int, double,...) without type checking.

```
// C program to illustrate macro function.
#include<stdio.h>
#define INC(P) ++P
int main()
{
    char *p = "Geeks";
    int x = 10;
    printf("%s ", INC(p));
    printf("%d", INC(x));
    return 0;
}
```

Output:

```
Geeks 11
```

Therefore we avoid to use Macro. But after the implementation of C11 standard in C programming, we can use Macro with the help of a new keyword i.e. "_Generic". We can define MACRO for the different types of data types. For example, the following macro INC(x) translates to INCl(x), INC(x) or INCf(x) depending on the type of x:

```
#define INC(x) _Generic((x), long double: INCl, \
                        default: INC, \
                        float: INCf)(x)
```



```

// C program to illustrate macro function.
#include <stdio.h>
int main(void)
{
    // _Generic keyword acts as a switch that chooses
    // operation based on data type of argument.
    printf("%d\n", _Generic( 1.0L, float:1, double:2,
                             long double:3, default:0));
    printf("%d\n", _Generic( 1L, float:1, double:2,
                             long double:3, default:0));
    printf("%d\n", _Generic( 1.0L, float:1, double:2,
                             long double:3));
    return 0;
}

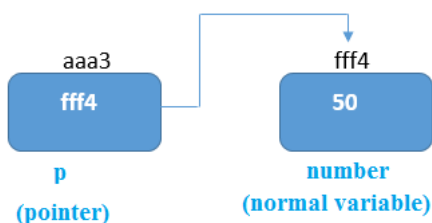
```

Output:

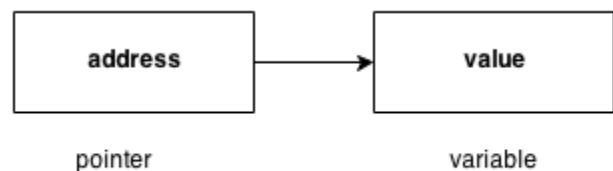
3
0
3

10. Pointers:

The pointer in C language is a variable which stores the address of another variable. This variable can be of type int, char, array, function, or any other pointer. The size of the pointer depends on the architecture. However, in 32-bit architecture the size of a pointer is 4 bytes.



javatpoint.com



Advantage of pointer

- 1) Pointer **reduces the code** and **improves the performance**, it is used to retrieving strings, trees, etc. and used with arrays, structures, and functions.
- 2) We can **return multiple values from a function** using the pointer.
- 3) It makes you able to **access any memory location** in the computer's memory.

There are many applications of pointers in c language.

1) Dynamic memory allocation

In c language, we can dynamically allocate memory using malloc() and calloc() functions where the pointer is used.

2) Arrays, Functions, and Structures

Pointers in c language are widely used in arrays, functions, and structures. It reduces the code and improves the performance.

a. Types:

There are eight different types of pointers which are as follows –

- ▣ Null pointer
- ▣ Void pointer
- ▣ Wild pointer
- ▣ Dangling pointer
- ▣ Complex pointer
- ▣ Near pointer
- ▣ Far pointer
- ▣ Huge pointer

Null Pointer

You create a null pointer by assigning the null value at the time of pointer declaration.

This method is useful when you do not assign any address to the pointer. A null pointer always contains value 0.

Example

Following is the C program for the null pointer –

```
#include <stdio.h>
int main(){
    int *ptr = NULL; //null pointer
    printf("The value inside variable ptr is:\n%d",ptr);
    return 0;
}
```

[Live Demo](#)

Output

When the above program is executed, it produces the following result –

```
The value inside variable ptr is:
0
```


Void Pointer

It is a pointer that has no associated data type with it. A void pointer can hold addresses of any type and can be typecast to any type.

It is also called a generic pointer and does not have any standard data type.

It is created by using the keyword void.

Example

Following is the C program for the void pointer –

[Live Demo](#)

```
#include <stdio.h>
int main(){
    void *p = NULL; //void pointer
    printf("The size of pointer is:%d\n",sizeof(p)); //size of p depends on architecture
    return 0;
}
```

Output

When the above program is executed, it produces the following result –

```
The size of pointer is:8
```

Wild Pointer

Wild pointers are also called uninitialized pointers. Because they point to some arbitrary memory location and may cause a program to crash or behave badly.

This type of C pointer is not efficient. Because they may point to some unknown memory location which may cause problems in our program. This may lead to the crashing of the program.

It is advised to be cautious while working with wild pointers.

Example

Following is the C program for the wild pointer –

```
#include <stdio.h>
int main(){
    int *p; //wild pointer
    printf("\n%d",*p);
    return 0;
}
```

Process returned -1073741819 (0xC0000005) execution time : 1.206 s
Press any key to continue
i.e. you won't get output, some compilers show error message at output

Dangling pointer

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

1. De-allocation of memory

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

2. Function Call

```
// The pointer pointing to local variable becomes
// dangling when local variable is not static.
#include<stdio.h>

int *fun()
{
    // x is local variable and goes out of
    // scope after an execution of fun() is
    // over.
    int x = 5;

    return &x;
}

// Driver Code
int main()
{
    int *p = fun();
    fflush(stdin);

    // p points to something which is not
    // valid anymore
    printf("%d", *p);
    return 0;
}
```

Output:

A garbage Address

The above problem doesn't appear (or p doesn't become dangling) if x is a static variable.

3. Variable goes out of scope

```
void main()
{
    int *ptr;
    .....
    .....
    {
        int ch;
        ptr = &ch;
    }
    .....
    // Here ptr is dangling pointer
}
```

Near Pointer:

- Near pointer means a pointer that is utilized to bit address of up to 16 bits within a given section of that computer memory which is 16 bit enabled.
- It can only access data of the small size of about 64 kb within a given period, which is the main disadvantage of this type of pointer.

Example:

```
#include<stdio.h>

int main()
{
    int a= 300;

    int near* ptr;

    ptr= &a;

    printf("%d",sizeof ptr);

    return 0;
}
```

Output: 3

Far Pointer:

- A far pointer is typically 32 bit which can access memory outside that current segment.
- To utilize the far pointer, the compiler allows a segment register to save segment address, then another register to save offset inside the current segment.

Example:

```
#include<stdio.h>

int main()
{
    int a= 10;

    int far *ptr;

    ptr=&a;

    print("%d", sizeof ptr);

    return 0;
}
```

Huge Pointer:

- Same as far pointer huge pointer is also typically 32 bit which can access outside the segment.
- A far pointer that is fixed and hence that part of that sector within which they are located cannot be changed in any way; huge pointers can be.

Example:

```
#include<stdio.h>

int main()
{
    char huge *far *a;

    printf("%d%d%d", sizeof(a), size(*a), sizeof(**a));

    return 0;
}
```

Output: 4 4 1.

b. Double Pointer:

We already know that a pointer points to a location in memory and thus used to store the address of variables. So, when we define a pointer to pointer. The first pointer is used to store the address of the variable. And the second pointer is used to store the address of the first pointer. That is why they are also known as double pointers.

How to declare a pointer to pointer in C?

Declaring Pointer to Pointer is similar to declaring pointer in C. The difference is we have to place an additional '*' before the name of pointer.

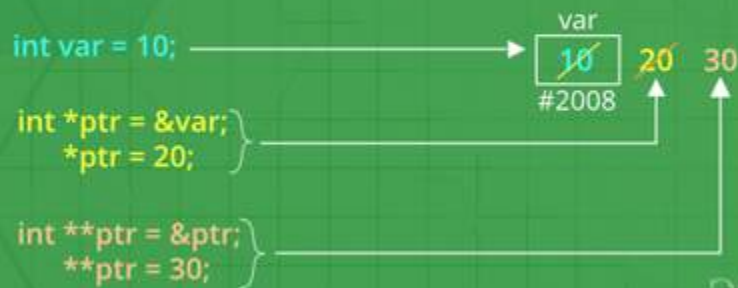
Syntax:

```
int **ptr;    // declaring double pointers
```

Double Pointer



How pointer works in C



c. Reference and dereference:

Referencing means taking the address of an existing variable (using &) to set a pointer variable. In order to be valid, a pointer has to be set to the address of a variable of the same type as the pointer, without the asterisk:

```
int c1;
int* p1;
c1 = 5;
p1 = &c1;
//p1 references c1
```

Dereferencing a pointer means using the * operator (asterisk character) to retrieve the value from the memory address that is pointed by the pointer: NOTE: The value stored at the address of the pointer must be a value OF THE SAME TYPE as the type of variable the pointer "points" to, but there is **no guarantee** this is the case unless the pointer was set correctly. The type of variable the pointer points to is the type less the outermost asterisk.

```
int n1;
n1 = *p1;
```

```
#include<iostream>
using namespace std;

void main()
{
    int x;
    int *ptr_p;

    x = 5;
    ptr_p = &x;

    cout << *ptr_p;
}
```

Note: If you forget to place * (in front of the pointer) in the cout statement, you will print the address of integer x. (Try it).

Reference and dereference operators

In the example above we used ampersand sign (&). This sign is called the reference operator. If the reference operator is used you will get the "address of" a variable. In the example above we said: `ptr_p = &x;`. In words: store the address of the variable x in the pointer ptr_p.

We also used the asterisk sign (*) in the cout statement. This sign is called the dereference operator. If the dereference operator is used you will get the "value pointed by" a pointer. So we said: `cout << *ptr_p;`. In words: print (or put into the stream) the value pointed by ptr_p. (It will print the contents of integer x.)

Invalid dereferencing may or may not cause crashes:

- Dereferencing an uninitialized pointer can cause a crash
- Dereferencing with an invalid type cast will have the potential to cause a crash.
- Dereferencing a pointer to a variable that was dynamically allocated and was subsequently de-allocated can cause a crash
- Dereferencing a pointer to a variable that has since gone out of scope can also cause a crash.

Invalid referencing is more likely to cause compiler errors than crashes, but it's not a good idea to rely on the compiler for this.

d. Pointer to Function:

So let us first see ..how do we declare a function? For example,

```
int foo(int);
```

Here foo is a function that returns int and takes one argument of int type. So as a logical guy will think, by putting a * operator between int and foo(int) should create a pointer to a function i.e.

```
int * foo(int);
```

But Oops..C operator precedence also plays role here ..so in this case, operator () will take priority over operator *. And the above declaration will mean – a function foo with one argument of int type and return value of int * i.e. integer pointer. So it did something that we didn't want to do. 😞

So as a next logical step, we have to bind operator * with foo somehow. And for this, we would change the default precedence of C operators using () operator.

```
int (*foo)(int);
```

That's it. Here * operator is with foo which is a function name. And it did the same that we wanted to do.

So that wasn't as difficult as we thought earlier!

Function Pointer:

In C, like [normal data pointers](#) (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;
}
```

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.

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;
}

```

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.

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.

e. Restrict keyword:

In the [C programming language](#) (after 99 standard), a new keyword is introduced known as restrict.

- restrict keyword is mainly used in pointer declarations as a type qualifier for pointers.
- It doesn't add any new functionality. It is only a way for programmer to inform about an optimizations that compiler can make.
- When we use restrict with a pointer ptr, it tells the compiler that ptr is the only way to access the object pointed by it and compiler doesn't need to add any additional checks.
- If a programmer uses restrict keyword and violate the above condition, result is undefined behavior.
- restrict is not supported by C++. It is a C only keyword.

```
// C program to use restrict keyword.
#include <stdio.h>

// Note that the purpose of restrict is to
// show only syntax. It doesn't change anything
// in output (or logic). It is just a way for
// programmer to tell compiler about an
// optimization
void use(int* a, int* b, int* restrict c)
{
    *a += *c;

    // Since c is restrict, compiler will
    // not reload value at address c in
    // its assembly code. Therefore generated
    // assembly code is optimized
    *b += *c;
}

int main(void)
{
    int a = 50, b = 60, c = 70;
    use(&a, &b, &c);
    printf("%d %d %d", a, b, c);
    return 0;
}
```

Output:

```
120 130 70
```

11. Storage class:

Storage classes in C				
Storage Specifier	Storage	Initial value	Scope	Life
auto	stack	Garbage	Within block	End of block
extern	Data segment	Zero	global Multiple files	Till end of program
static	Data segment	Zero	Within block	Till end of program
register	CPU Register	Garbage	Within block	End of block

a. Volatile:

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

Objects declared as volatile are omitted from optimization because their values can be changed by code outside the scope of current code at any time. The system always reads the current value of a volatile object from the memory location rather than keeping its value in temporary register at the point it is requested, even if a previous instruction asked for a value from the same object. So the simple question is, how can value of a variable change in such a way that compiler cannot predict. Consider the following cases for answer to this question.

1) Global variables modified by an interrupt service routine outside the scope: For example, a global variable can represent a data port (usually global pointer referred as memory mapped IO) which will be updated dynamically. The code reading data port must be declared as volatile in order to fetch latest data available at the port. Failing to declare variable as volatile, the compiler will optimize the code in such a way that it will read the port only once and keeps using the same value in a temporary register to speed up the program (speed optimization). In general, an ISR used to update these data port when there is an interrupt due to availability of new data

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

If we do not use volatile qualifier, the following problems may arise

- 1) Code may not work as expected when optimization is turned on.
- 2) Code may not work as expected when interrupts are enabled and used.

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

12. Enum, struct and Union:

- The structure is a user-defined data type that is available in C++.
- Structures are used to combine different types of data types, just like an array is used to combine the same type of data types.
- A structure is declared by using the keyword "**struct**". When we declare a variable of the structure we need to write the keyword "**struct**" in C language but for C++ the keyword is not mandatory

Unions: A union is a type of structure that can be used where the amount of memory used is a key factor.

- Similarly to the structure, the union can contain different types of data types.
- Each time a new variable is initialized from the union it overwrites the previous in C language but in C++ we also don't need this keyword and uses that memory location.
- This is most useful when the type of data being passed through functions is unknown, using a union which contains all possible data types can remedy this problem.
- It is declared by using the keyword "**union**".

Enums: Enums are user-defined types that consist of named integral constants.

- It helps to assign constants to a set of names to make the program easier to read, maintain and understand.
- An Enumeration is declared by using the keyword "**enum**".

With a union, you're only supposed to use one of the elements, because they're all stored at the same spot. This makes it useful when you want to store something that could be one of several types. A struct, on the other hand, has a separate memory location for each of its elements and they all can be used at once.

To give a concrete example of their use, I was working on a Scheme interpreter a little while ago and I was essentially overlaying the Scheme data types onto the C data types. This involved storing in a struct an enum indicating the type of value and a union to store that value.

```

union foo {
    int a; // can't use both a and b at once
    char b;
} foo;

struct bar {
    int a; // can use both a and b simultaneously
    char b;
} bar;

union foo x;
x.a = 3; // OK
x.b = 'c'; // NO! this affects the value of x.a!

struct bar y;
y.a = 3; // OK
y.b = 'c'; // OK

```

edit: If you're wondering what setting x.b to 'c' changes the value of x.a to, technically speaking it's undefined. On most modern machines a char is 1 byte and an int is 4 bytes, so giving x.b the value 'c' also gives the first byte of x.a that same value:

```

union foo x;
x.a = 3;
x.b = 'c';
printf("%i, %i\n", x.a, x.b);

```

prints

```

99, 99

```

Why are the two values the same? Because the last 3 bytes of the int 3 are all zero, so it's also read as 99. If we put in a larger number for x.a, you'll see that this is not always the case:

```
union foo x;
x.a = 387439;
x.b = 'c';
printf("%i, %i\n", x.a, x.b);
```

prints

```
387427, 99
```

To get a closer look at the actual memory values, let's set and print out the values in hex:

```
union foo x;
x.a = 0xDEADBEEF;
x.b = 0x22;
printf("%x, %x\n", x.a, x.b);
```

prints

```
deadbe22, 22
```

You can clearly see where the 0x22 overwrote the 0xEF.

BUT

In C, the order of bytes in an int are **not defined**. This program overwrote the 0xEF with 0x22 on my Mac, but there are other platforms where it would overwrite the 0xDE instead because the order of the bytes that make up the int were reversed. Therefore, when writing a program, you should never rely on the behavior of overwriting specific data in a union because it's not portable.

For more reading on the ordering of bytes, check out [endianness](#).

[Endianness](#)

13. Bit Fields:

Suppose your C program contains a number of TRUE/FALSE variables grouped in a structure called status, as follows –

```
struct {  
    unsigned int widthValidated;  
    unsigned int heightValidated;  
} status;
```

This structure requires 8 bytes of memory space but in actual, we are going to store either 0 or 1 in each of the variables. The C programming language offers a better way to utilize the memory space in such situations.

If you are using such variables inside a structure then you can define the width of a variable which tells the C compiler that you are going to use only those number of bytes. For example, the above structure can be re-written as follows –

```
struct {  
    unsigned int widthValidated : 1;  
    unsigned int heightValidated : 1;  
} status;
```

The above structure requires 4 bytes of memory space for status variable, but only 2 bits will be used to store the values.

If you will use up to 32 variables each one with a width of 1 bit, then also the status structure will use 4 bytes. However as soon as you have 33 variables, it will allocate the next slot of the memory and it will start using 8 bytes. Let us check the following example to understand the concept –

Live Demo

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

/* define simple structure */
struct {
    unsigned int widthValidated;
    unsigned int heightValidated;
} status1;

/* define a structure with bit fields */
struct {
    unsigned int widthValidated : 1;
    unsigned int heightValidated : 1;
} status2;

int main( ) {
    printf( "Memory size occupied by status1 : %d\n", sizeof(status1));
    printf( "Memory size occupied by status2 : %d\n", sizeof(status2));
    return 0;
}
```

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

```
Memory size occupied by status1 : 8
Memory size occupied by status2 : 4
```

Bit Field Declaration

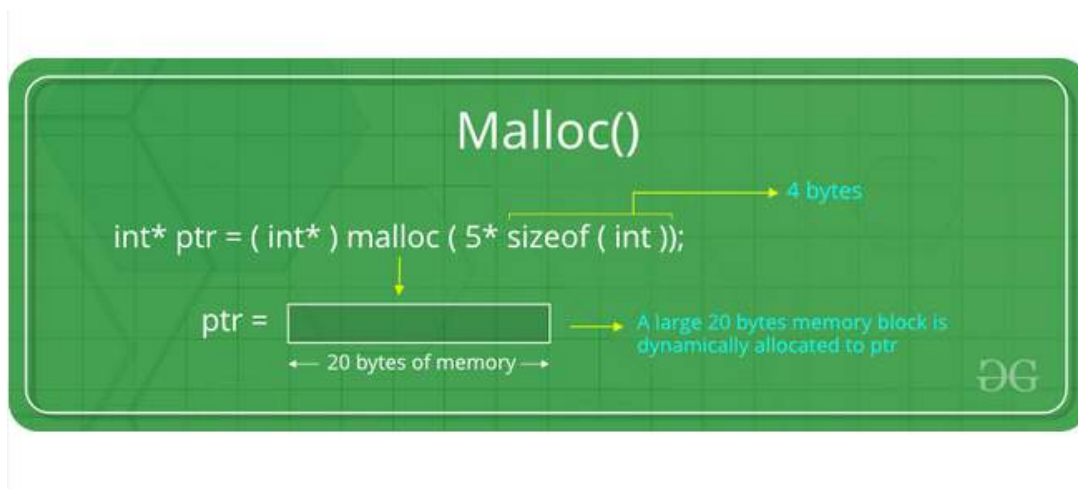
The declaration of a bit-field has the following form inside a structure –

```
struct {
    type [member_name] : width ;
};
```

The following table describes the variable elements of a bit field –

Sr.No.	Element & Description
1	type An integer type that determines how a bit-field's value is interpreted. The type may be int, signed int, or unsigned int.
2	member_name The name of the bit-field.
3	width The number of bits in the bit-field. The width must be less than or equal to the bit width of the specified type.

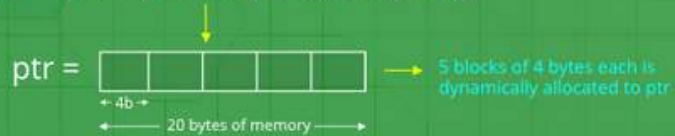
14. Malloc and Calloc:



If space is insufficient, allocation fails and returns a NULL pointer.

Calloc()

```
int* ptr = ( int* ) calloc ( 5, sizeof ( int ));
```



If space is insufficient, allocation fails and returns a NULL pointer.

Realloc()

```
int* ptr = ( int* ) malloc ( 5* sizeof ( int ));
```

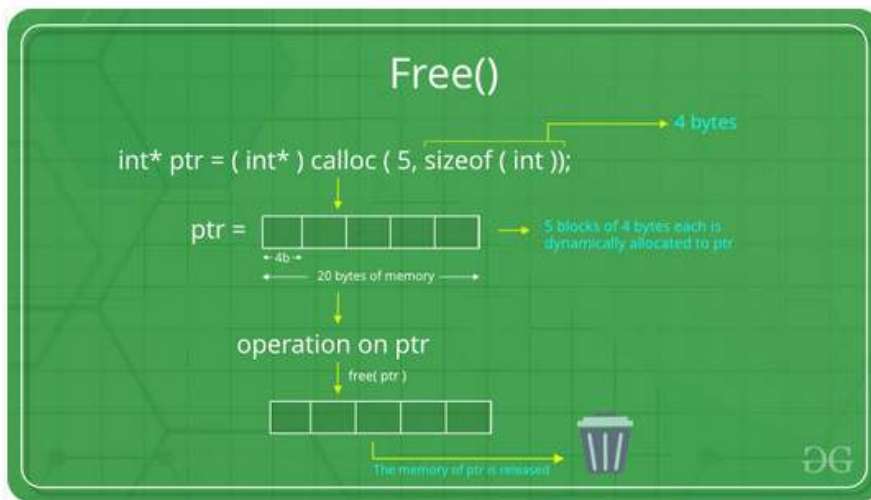


```
ptr = realloc ( ptr, 10* sizeof ( int ));
```



If space is insufficient, allocation fails and returns a NULL pointer.

15. Deallocation of memory:
a. With free:



- b. Without free:

Solution: Standard library function `realloc()` can be used to deallocate previously allocated memory. Below is function declaration of "realloc()" from "stdlib.h"

`void *realloc(void *ptr, size_t size);`



16. Precedence Table:

OPERATOR	TYPE	ASSOCIATIVITY
() [] . ->		left-to-right
++ -- +- ! ~ (type) * & sizeof	Unary Operator	right-to-left
* / %	Arithmetic Operator	left-to-right
+ -	Arithmetic Operator	left-to-right
<< >>	Shift Operator	left-to-right
< <= > >=	Relational Operator	left-to-right
== !=	Relational Operator	left-to-right
&	Bitwise AND Operator	left-to-right
^	Bitwise EX-OR Operator	left-to-right
	Bitwise OR Operator	left-to-right
&&	Logical AND Operator	left-to-right
	Logical OR Operator	left-to-right
? :	Ternary Conditional Operator	right-to-left
= += -= *= /= %= &= ^= = <<= >>=	Assignment Operator	right-to-left
,	Comma	left-to-right

17. File Handling:

No.	Function	Description
1	fopen()	opens new or existing file
2	fprintf()	write data into the file
3	fscanf()	reads data from the file
4	fputc()	writes a character into the file
5	fgetc()	reads a character from file
6	fclose()	closes the file
7	fseek()	sets the file pointer to given position
8	fputw()	writes an integer to file
9	fgetw()	reads an integer from file
10	ftell()	returns current position
11	rewind()	sets the file pointer to the beginning of the file

18. Preprocessor Directive:

The **C Preprocessor** is not a part of the compiler, but is a separate step in the compilation process. In simple terms, a C Preprocessor is just a text substitution tool and it instructs the compiler to do required pre-processing before the actual compilation. We'll refer to the C Preprocessor as CPP.

All preprocessor commands begin with a hash symbol (#). It must be the first nonblank character, and for readability, a preprocessor directive should begin in the first column. The following section lists down all the important preprocessor directives –

Sr.No.	Directive & Description
1	#define Substitutes a preprocessor macro.
2	#include Inserts a particular header from another file.
3	#undef Undefines a preprocessor macro.
4	#ifdef Returns true if this macro is defined.
5	#ifndef Returns true if this macro is not defined.
6	#if Tests if a compile time condition is true.
7	#else The alternative for #if.
8	#elif else and #if in one statement.
9	#endif Ends preprocessor conditional.
10	#error Prints error message on stderr.
11	#pragma Issues special commands to the compiler, using a standardized method.

The Stringize (#) Operator

The stringize or number-sign operator ('#'), when used within a macro definition, converts a macro parameter into a string constant. This operator may be used only in a macro having a specified argument or parameter list. For example –

```
#include <stdio.h>

#define message_for(a, b) \
    printf("#a " and " #b ": We love you!\n")

int main(void) {
    message_for(Carole, Debra);
    return 0;
}
```

[Live Demo](#)

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

```
Carole and Debra: We love you!
```

[More](#)

19. Multithreading:

What is a Thread?

A thread is a single sequence stream within in a process. Because threads have some of the properties of processes, they are sometimes called *lightweight processes*.

What are the differences between process and thread?

Threads are not independent of one other like processes as a result threads shares with other threads their code section, data section and OS resources like open files and signals. But, like process, a thread has its own program counter (PC), a register set, and a stack space.

Why Multithreading?

Threads are popular way to improve application through parallelism. For example, in a browser, multiple tabs can be different threads. MS word uses multiple threads, one thread to format the text, other thread to process inputs, etc.

Threads operate faster than processes due to following reasons:

- 1) Thread creation is much faster.
- 2) Context switching between threads is much faster.
- 3) Threads can be terminated easily
- 4) Communication between threads is faster.

Can we write multithreading programs in C?

Unlike Java, multithreading is not supported by the language standard.

POSIX Threads, usually referred to as **pthread**, is an [execution model](#) that exists independently from a language, as well as a parallel execution model. It allows a program to control multiple different flows of work that overlap in time. Each flow of work is referred to as a [thread](#), and creation and control over these flows is achieved by making calls to the POSIX Threads API. [POSIX Threads](#) is an [API](#) defined by the standard *POSIX.1c, Threads extensions (IEEE Std 1003.1c-1995)*.

A simple C program to demonstrate use of pthread basic functions

Please note that the below program may compile only with C compilers with pthread library.

```

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h> //Header file for sleep(). man 3 sleep for details.
#include <pthread.h>

// A normal C function that is executed as a thread
// when its name is specified in pthread_create()
void *myThreadFun(void *vargp)
{
    sleep(1);
    printf("Printing GeeksQuiz from Thread \n");
    return NULL;
}

int main()
{
    pthread_t thread_id;
    printf("Before Thread\n");
    pthread_create(&thread_id, NULL, myThreadFun, NULL);
    pthread_join(thread_id, NULL);
    printf("After Thread\n");
    exit(0);
}

```

20. Assertion:

Assertion Vs Normal Error Handling

Assertions are mainly used to check logically impossible situations. For example, they can be used to check the state a code expects before it starts running or state after it finishes running. Unlike normal error handling, assertions are generally disabled at run-time. Therefore, it is not a good idea to write statements in `assert()` that can cause side effects. For example writing something like `assert(x = 5)` is not a good idea as `x` is changed and this change won't happen when assertions are disabled. See [this](#) for more details.

Ignoring Assertions

In C/C++, we can completely remove assertions at compile time using the preprocessor `NDEBUG`.

21. Hygienic Macros:

A **hygienic macro** is a macro that defines variables for its own use without accidentally confusing them with the variables defined by the user of the macro. By design, C/C++ preprocessor macro performs only simple string substitution which is not hygienic. That means those using a macro has to be mindful of the macro's definition and avoid using variables of already used names. This also means that such macro cannot be arbitrarily nested in scope, since nesting will cause variable conflict.

But hygienic macro is really useful for defining language extensions in the form of syntactic sugars. My motivation example is a foreach loop that was not added to the C++ language until C++11. Although C++ now has a ranged-for syntax, it is still useful for a plain C library that implements containers. There may be other use cases for macro hygiene, so I want to explain the technique I have been using for a number of years.

If we see the following code, we can see that it is not working properly.

Example

```
#include<stdio.h>
#define INCREMENT(i) do { int a = 0; ++i; } while(0)
main(void) {
    int a = 10, b = 20;
    //Call the macros two times for a and b
    INCREMENT(a);
    INCREMENT(b);
    printf("a = %d, b = %d\n", a, b);
}
```

After preprocessing the code will be like this –

Example

```
#include<stdio.h>
#define INCREMENT(i) do { int a = 0; ++i; } while(0)
main(void) {
    int a = 10, b = 20;
    //Call the macros two times for a and b
    do { int a = 0; ++a; } while(0) ;
    do { int a = 0; ++b; } while(0) ;
    printf("a = %d, b = %d\n", a, b);
}
```

Output

```
a = 10, b = 21
```

Here we can see the value is not updated for a. So in this case we will use the hygienic macros. These Hygienic macros are macros whose expansion is guaranteed that it does not create the accidental capture of identifiers. Here we will not use any variable name that can risk interfacing with the code under expansion. Here another variable 't' is used inside the macro. This is not used in the program itself.

Example

```
#include<stdio.h>
#define INCREMENT(i) do { int t = 0; ++i; } while(0)
main(void) {
    int a = 10, b = 20;
    //Call the macros two times for a and b
    INCREMENT(a);
    INCREMENT(b);
    printf("a = %d, b = %d\n", a, b);
}
```

Output

```
a = 11, b = 21
```