

SALE!  
GeeksforGeeks Courses Upto 25% Off Enroll Now!

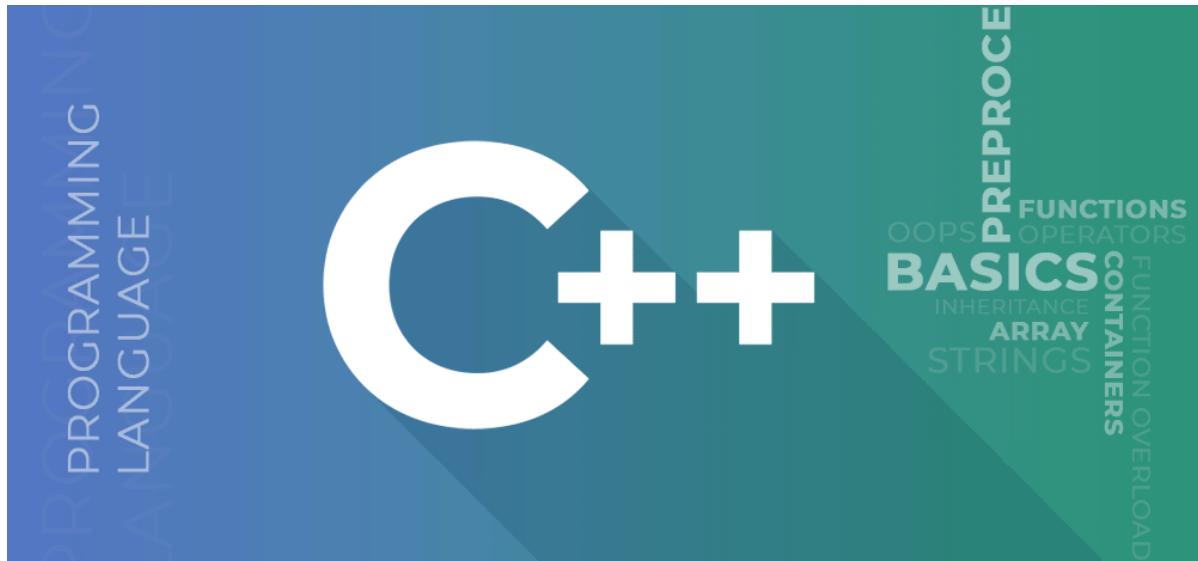
[Save 25% on Courses](#)[DSA](#)[Data Structures](#)[Algorithms](#)[Interview Preparation](#)[Data Science](#)[T](#)

# C++ Programming Language

Last Updated : 05 Mar, 2023

[Read](#)[Discuss\(250+\)](#)[Courses](#)[Practice](#)[Video](#)

C++ is a general-purpose programming language and is widely used nowadays for competitive programming. It has imperative, object-oriented and generic programming features. C++ runs on lots of platforms like Windows, Linux, Unix, Mac etc.

[C++ Recent Articles!](#)[C++ Interview Questions](#)[C++ Programs](#)

Basics, C vs C++, C++ vs Java, Input and Output, Operators, Arrays and Strings, Functions, References and Pointers, Dynamic memory allocation, Object Oriented Programming(OOP),Constructor and Destructor, Function Overloading, Operator Overloading, Virtual Functions, Exception Handling, Namespaces, Standard Template Library (STL), Inheritance, C++ Library, C++ Advanced, C++ in Competitive Programming, Puzzles, Interview Questions, Multiple Choice Questions

## Basics

1. Setting up C++  
Development Environment
2. Writing first C++ program(Practice)

## Standard Template Library (STL)

### Algorithms

1. Introduction to STL
2. Sorting

3. void main or main()
4. C++ Data Types(Practice)
5. Basic Input/Output
6. Response on exceeding valid range of data types
7. C++ Preprocessors
8. Operators in C++(Practice)
9. Loops (Practice)
10. Decision Making in C++(Practice)
11. Execute both if and else simultaneously
12. How to compile 32-bit program on 64-bit gcc in C and C++
13. Switch statement in C++(Practice)
14. Functions in C++(Practice)
15. Arrays in C/C++(Practice)
16. Strings in C++(Practice)
17. Pointers in C++(Practice)
18. References in C++
19. Introduction to OOP in C++

### C vs C++

1. C program that won't compile in C++
2. Undefined Behaviour in C and C++
3. Name Mangling and extern "C" in C++
4. void \* in C vs C++
5. Program that produces different results in C and C++
6. Type difference of character literals in C vs C++
7. Difference between Structures in C and C++

### C++ vs Java

1. Inheritance in C++ vs Java
2. static keyword in C++ vs Java
3. default virtual behavior in C++ vs Java
4. Exception Handling in C++ vs Java
5. Foreach in C++ vs Java
6. Templates in C++ vs Generics in Java
7. Floating Point Operations & Associativity in C, C++ and Java
8. Similarities between Java and C++

### Input and output

### 3. Searching

#### Containers:

1. Pair (Practice)
2. Vector (Practice)
  - Ways to copy a vector in C++
  - Sorting 2D Vector in C++ | Set 3 (By number of columns), (Sort in descending order by first and second)
  - Sorting 2D Vector in C++ | Set 2 (In descending order by row and column)
  - 2D vector in C++ with user defined size
  - Vector::clear() and vector::erase() in C++ STL
  - Passing vector to a function in C++
  - Vector::push\_back() and vector::pop\_back() in C++ STL
  - Vector::empty() and vector::size() in C++ STL
  - vector::front() and vector::back() in C++ STL
  - Initialize a vector; Different ways
  - Sorting 2D Vector in C++ | Set 1 (By row and column), (Sort by first and second)
  - Computing index using pointers returned by STL functions in C++
3. List
  - List in C++ | Set 2 (Some Useful Functions)
  - Forward List in C++ | Set 1 (Introduction and Important Functions)
  - Forward List in C++ | Set 2 (Manipulating Functions)
  - list::remove() and list::remove\_if() in C++ STL
  - Forward\_list::front() and forward\_list::empty() in C++ STL
  - Forward\_list::remove() and forward\_list::remove\_if() in C++ STL
  - forward\_list::unique() in C++ STL
  - forward\_list::reverse() in C++ STL
  - forward\_list::max\_size() in C++ STL
  - forward\_list::before\_begin() in C++ STL

1. I/O Redirection in C++
2. Clearing The Input Buffer
3. Basic Input/Output(Practice)
4. `cout << endl` vs `cout << "\n"` in C++
5. Problem with `scanf()` when there is `fgets()/gets()/scanf()` after it
6. How to use `getline()` in C++ when there are blank lines in input?
7. `scanf()` and `fscanf()` in C – Simple Yet Powerful
8. Using return value of `cin` to take unknown number of inputs in C++
9. How to change the output of `printf()` in `main()` ?
10. Implementation of a Falling Matrix
11. What does buffer flush means in C++ ?
12. `kbhit` in C language
13. Code to generate the map of India

## Operators

1. Operators in C++
2. Unary operators in C/C++
3. Conditionally assign a value without using conditional and arithmetic operators
4. Execution of `printf` with `++` operators
5. Set a variable without using Arithmetic, Relational or Conditional Operator
6. Scope Resolution Operator vs this pointer
7. Pre-increment (or pre-decrement)
8. `new` and `delete` operator in C++
9. CHAR\_BIT in C
10. Casting operators | Set 1 (`const_cast`)

## Arrays and Strings

1. Arrays in C/C++
2. Array of Strings
3. Multidimensional arrays in C/C++
4. Raw string literal
5. Counts of distinct consecutive sub-string of length two
6. Converting string to number and vice-versa
7. Find size of array in C/C++ without using `sizeof`

- `forward_list::cbefore_begin()` in C++ STL
- `forward_list::unique()` in C++ STL
- `forward_list::before_begin()` in C++ STL
- `forward_list::cbefore_begin()` in C++ STL
- `forward_list::reverse()` in C++ STL
- `forward_list::max_size()` in C++ STL
- `forward_list::splice_after()` in C++ STL
- `list::empty()` and `list::size()` in C++ STL
- `list::front()` and `list::back()` in C++ STL
- `list::pop_front()` and `list::pop_back()` in C++ STL
- `list::push_front()` and `list::push_back()` in C++ STL
- `list push_front()` function in C++ STL
- `list pop_back()` function in C++ STL
- `list pop_front()` function in C++ STL
- `list reverse` function in C++ STL
- `list resize()` function in C++ STL
- `list size()` function in C++ STL
- `list max_size()` function in C++ STL

4. Dequeue
5. `Deque::empty()` and `deque::size()` in C++ STL
6. `Deque::pop_front()` and `deque::pop_back()` in C++ STL
7. `Deque::clear()` and `deque::erase()` in C++ STL
8. Queue (Practice)
9. `Queue::front()` and `queue::back()` in C++ STL
- 10.
11. `Queue::push()` and `queue::pop()` in C++ STL
12. `queue::empty()` and `queue::size()` in C++ STL
13. Priority Queue
14. Stack (Practice)
15. `Stack::push()` and `stack::pop()` in C++ STL
16. `Forward_list :: push_front()` and `forward_list :: pop_front()` in C++ STL
17. `Stack::top()` in C++ STL
18. `Stack::empty()` and `stack::size()` in C++ STL
19. Set (Practice)
  - Count number of unique Triangles using STL | Set 1 (Using set)
  - `std::istream_iterator` and `std::ostream_iterator` in C++ STL
20. `Std::next_permutation` and `prev_permutation` in C++

8. How to quickly reverse a string in C++?
9. Tokenizing a string in C++
10. Getline() function and character array
11. Convert string to char array in C++
12. C++ string class and its applications , Set 2
13. How to create a dynamic 2D array inside a class in C++ ?
14. Lexicographically next permutation
15. Print size of array parameter
16. Split a string in C/C++, Python and Java
17. Stringstream in C++ and its applications
18. Strchr() function in C/C++
19. Iisspace() in C/C++ and its application to count whitespace characters
20. Char\* vs std::string vs char[] in C++
21. Std::lexicographical\_compare() in C++STL
22. Std::string::at in C++
23. Std::substr() in C/C++
24. std::stol() and std::stoll() functions in C++
25. Extract all integers from string in C++
26. Strchr() function in C++ and its applications
27. Strcat() vs strncat() in C++
28. Strncat() function in C/C++
29. Strpbrk() in C
30. strcoll() in C/C++
31. Why strcpy and strncpy are not safe to use?
21. Std::stoul and std::stoull in C++
22. Shuffle vs random\_shuffle in C++
23. Difference between set, multiset, unordered\_set, unordered\_multiset
24. Check if a key is present in a C++ map or unordered\_map
25. Std::stable\_partition in C++
26. Valarray slice selector
27. Std::memchr in C++
28. Std::strncmp() in C++
29. Stable\_sort() in C++ STL
30. Std::memcmp() in C++
31. Std::memset in C++
32. Std::bucket\_count and std::bucket\_size in unordered\_map in C++
33. Map of pairs in STL
34. Range-based for loop in C++
35. Std::includes() in C++ STL
36. Std::set\_symmetric\_difference in C++
37. Std::sort\_heap in C++
38. Map vs unordered\_map in C++
39. Round() in C++
40. Modulus of two float or double numbers
41. Multiset
42. Map (Practice)
43. Heap using STL C++

## Functions

1. Functions in C++
2. Default Arguments
3. C function argument and return values
4. Inline Functions
5. Return from void functions
6. Returning multiple values from a function using Tuple and Pair
7. Function Call Puzzle
8. Functors
9. Ciel and floor functions in C++
10. Const member functions
11. atol(), atoll() and atof() functions in C/C++
12. swap() in C++
13. wmemmove() function in c++
14. wcscat() function in C++
15. wcsncmp() function in C++ with Examples

## Multimap

- Multimap in C++ Standard Template Library (STL)
- multimap::find() in C++ STL
- multimap::erase() in C++ STL
- map emplace() in C++ STL
- multimap::emplace\_hint() in C++ STL
- multimap::emplace() in C++ STL
- multimap::count() in C++ STL
- multimap::find() in C++ STL
- multimap::erase() in C++ STL
- multimap::begin() and multimap::end() in C++ STL
- multimap::cbegin() and multimap::cend() in C++ STL
- map cbegin() and cend() function in C++ STL

- 16. [wcscpy\(\) function in C++ with Examples](#)
- 17. [wcslen\(\) function in C++ with Examples](#)
- 18. [difftime\(\) function in C++](#)
- 19. [asctime\(\) function in C++](#)
- 20. [localtime\(\) function in C++](#)
- 21. [scalbn\(\) function in C++](#)
- 22. [isunordered\(\) function in C++](#)
- 23. [isnormal\(\) in C++](#)
- 24. [isinf\(\) function in C++](#)
- 25. [quick\\_exit\(\) function in C++ with Examples](#)
- 26. [ctime\(\) Function in C/C++](#)
- 27. [clock\(\) function in C/C++](#)
- 28. [nearbyint\(\) function in C++](#)
- 29. [quick\\_exit\(\) function in C++ with Examples](#)
- 30. [wcscmp\(\) function in C++ with Examples](#)
- 31. [wcscpy\(\) function in C++ with Examples](#)
- 32. [wcslen\(\) function in C++ with Examples](#)

## Pointers and References

- 1. [Pointers in C and C++](#)
- 2. [What is Array Decay in C++? How can it be prevented?](#)
- 3. [Opaque Pointer](#)
- 4. [References](#)
- 5. [Can references refer to invalid location?](#)
- 6. [Pass arguments by reference or pointer](#)
- 7. [Smart Pointers](#)
- 8. ['this' pointer](#)
- 9. [Type of 'this' pointer](#)
- 10. ["delete this"](#)
- 11. [auto\\_ptr, unique\\_ptr, shared\\_ptr and weak\\_ptr](#)
- 12. [Dangling, Void , Null and Wild Pointers](#)
- 13. [Passing by pointer Vs Passing by Reference](#)
- 14. [NaN in C++ – What is it and how to check for it?](#)
- 15. [nullptr](#)
- 16. [Pointers vs References in C++](#)

## Dynamic memory allocation

- 1. [new and delete operator in C++](#)
- 2. [malloc\(\) vs new](#)
- 3. [delete\(\) and free\(\)](#)
- 4. [Std::get\\_temporary\\_buffer in C++](#)

- [multimap::crbegin\(\) and multimap::crend\(\) in C++ STL](#)
- [multimap size\(\) function in C++ STL](#)
- [multimap lower\\_bound\(\) function in C++ STL](#)
- [multimap swap\(\) function in C++ STL](#)
- [multimap upper\\_bound\(\) function in C++ STL](#)
- [multimap maxsize\(\) in C++ STL](#)
- [multimap insert\(\) in C++ STL](#)
- [multimap equal\\_range\(\) in C++ STL](#)

## CPP-Math

- [sinh\(\) function in C++ STL](#)
- [cosh\(\) function in C++ STL](#)
- [tanh\(\) function in C++ STL](#)
- [acos\(\) function in C++ STL](#)
- [asinh\(\) function in C++ STL](#)
- [acosh\(\) function in C++ STL](#)
- [atanh\(\) function in C++ STL](#)

## More:

- 1. [sort\(\) in C++ STL](#)
- 2. [Strand sort](#)
- 3. [Type Inference in C++ \(auto and decltype\)](#)
- 4. [transform\(\) in C++ STL](#)
- 5. [Variadic function templates in C++](#)
- 6. [Template Specialization](#)
- 7. [Implementing iterator pattern of a singly linked list](#)
- 8. [Binary Search functions in C++ STL](#)
- 9. [Descending order in Map and Multimap of C++ STL](#)
- 10. [Insertion and Deletion in STL Set C++](#)
- 11. [set::key\\_comp\(\) in C++ STL](#)
- 12. [set value\\_comp\(\) function in C++ STL](#)
- 13. [unordered\\_set get\\_allocator\(\) in C++ STL with Examples](#)

## Inheritance

- [What all is inherited from parent class in C++?](#)
- [Virtual Functions and Runtime Polymorphism in C++](#)
- [Multiple Inheritance in C++](#)

## Object Oriented Programming(OOP)

1. Object oriented design
2. Introduction to OOP in C++
3. Classes and Objects
4. Access Modifiers
5. Inheritance
6. Polymorphism
7. Encapsulation
8. Data Abstraction
9. Structure vs class
10. Can a C++ class have an object of self type?
11. Why is the size of an empty class not zero?
12. Static data members in C++
13. Some interesting facts about static member functions
14. Friend class and function
15. Local Class
16. Nested Classes
17. Simulating final class

## Constructor and Destructor

1. Constructors
2. Copy Constructor
3. Destructors
4. Does compiler create default constructor when we write our own?
5. When should we write our own copy constructor?
6. When is copy constructor called?
7. Initialization of data members
8. Use of explicit keyword
9. When do we use Initializer List in?
10. Default Constructors
11. Private Destructor
12. Playing with Destructors
13. Copy elision
14. C++ default constructor | Built-in types
15. When does compiler create a default constructor and copy constructor?
16. Why copy constructor argument should be const in C++?
17. Advanced C++ | Virtual Constructor
18. Advanced C++ | Virtual Copy Constructor

- What happens when more restrictive access is given to a derived class method in C++?
- Object Slicing in C++
- Hiding of all overloaded methods in base class
- Inheritance and friendship
- Simulating final class

## C++ Library

1. <random> file – generators and distributions
2. Array type manipulation
3. C++ programming and STL facts
4. Sqrt, sqrtl and sqrtf in C++
5. std::stod, std::stof, std::stold in C++
6. C program to demonstrate fork() and pipe()
7. Complex numbers in C++ | Set 1 Set 2
8. Inbuilt library functions for user Input
9. Rename function in C/C++
10. Chrono
11. valarray class
12. Floating Point Manipulation (fmod(), remainder(), remquo() ... in cmath) (Practice)
13. Character Classification: cctype
14. Snprintf() in C library
15. Boost::split in C++ library
16. Modulus of two float or double numbers
17. Is\_trivial function in C++
18. Array sum in C++ STL
19. Div() function in C++
20. Exit() vs \_Exit() in C and C++
21. Std::none\_of in C++
22. Isprint() in C++
23. Iscntrl() in C++ and its application to find control characters
24. Std::partition\_point in C++
25. Iterator Invalidiation in C++
26. Fesetround() and fegetround() in C++ and their application
27. Rint(), rintf(), rintl() in C++
28. Hypot(), hypotf(), hypotl() in C++
29. Std::gslice | Valarray generalized slice selector
30. std::setbase, std::setw , std::setfill in C++
31. Strxfrm() in C/C++

19. When are static objects destroyed?

20. Is it possible to call constructor and destructor explicitly?

### Function Overloading

1. Function Overloading

2. Functions that can't be overloaded

3. Function overloading and const keyword

4. Function overloading and return type

5. Does overloading work with Inheritance?

6. Can main() be overloaded

7. Function Overloading and float

### Operator Overloading

1. Operator Overloading

2. Copy constructor vs assignment operator

3. When should we write our

own assignment operator?

4. Operators that cannot be overloaded

5. Conversion Operators

6. Is assignment operator inherited?

7. Default Assignment Operator and References

8. Overloading stream insertion (<<)  
and extraction (>>) operators

9. Overloading array index operator []

### Virtual Functions

1. Virtual Functions and Runtime Polymorphism

2. Default arguments and virtual function

3. Virtual functions in derived classes

4. Can static functions be virtual?

5. Virtual Destructor

6. Virtual Constructor

7. Virtual Copy Constructor

8. RTTI (Run-time type information)

9. Can virtual functions be private?

10. Inline virtual function

11. Pure Virtual Functions and Abstract Classes

12. Pure virtual destructor

### Exception Handling

1. Exception Handling Basics

2. Stack Unwinding

32. Set position with seekg() in C++ language file handling

33. Strstr() in C/C++

34. Difftime() C library function

35. Socket Programming

36. Precision of floating point numbers in C++  
(floor(), ceil(), trunc(), round() and  
setprecision())

37. <bit/stdc++.h> header file

38. std::string class in C++

39. Merge operations using STL in C++ (merge,  
includes, set\_union, set\_intersection,  
set\_difference, ..)

40. std::partition in C++ STL

41. Ratio Manipulations in C++ | Set 1  
(Arithmetic) , Set 2 (Comparison)

42. numeric header in C++ STL | Set 1 (accumulate()  
and partial\_sum()), Set 2  
(adjacent\_difference(), inner\_product() and  
iota())

43. Bind function and placeholders

44. Array class

45. Tuples

46. Regex (Regular Expression)

47. Common Subtleties in Vector STLs

48. Understanding constexpr specifier

49. unordered\_multiset and its uses

50. unordered\_multimap and its application

51. Populating a vector in C++ using fill() and  
fill\_n()

52. Writing OS Independent Code in C/C++

53. C Program to display hostname and IP  
address

54. Database Connectivity using C/C++

55. C++ bitset and its application

56. unordered\_map in STL and its applications

57. unordered\_set in STL and its applications

58. nextafter() and nexttoward()

### C++ Advanced

1. User Defined Literal

2. Placement new operator

3. Advanced C++ with boost library

4. Copy-and-Swap Idiom

3. Catching base and derived classes as exceptions
4. Catch block and type conversion
5. Exception handling and object destruction

### Namespace

1. Namespace in C++ | Set 1 (Introduction)
2. Set 2 (Extending namespace and Unnamed namespace)
3. Namespace in C++ | Set 3 (Accessing, creating header, nesting and aliasing)
4. Inline namespaces and usage of the "using" directive inside namespaces
5. Can namespaces be nested?

5. Zombie and Orphan Processes
6. Lambda expression
7. C++ | Signal Handling
8. Preventing Object Copy in C++
9. Command line arguments in C++

### C++ in Competitive Programming

1. Writing C/C++ code efficiently in Competitive programming
2. Useful Array algorithms in C++ STL
3. searching in fork()
4. Data Type Ranges and their macros
5. Cin-Cout vs Scanf-Printf
6. getchar\_unlocked() – faster input in C/C++ for Competitive Programming
7. C qsort() vs C++ sort()
8. Middle of three using minimum comparisons
9. Check for integer overflow on multiplication
10. Generating Test Cases (generate() and generate\_n())

### Puzzles

1. Can we call an undeclared function in?
2. Can we access global variable if there is a local variable with same name?
3. Can we use function on left side of an expression in C and C++?
4. Can we access private data members of a class without using a member or a friend function?
5. How to make a C++ class whose objects can only be dynamically allocated?
6. How to print "GeeksforGeeks" with empty main()
7. Print 1 to 100, without loop and recursion
8. C/C++ Tricky Programs
9. Print a number 100 times without using loop, recursion and macro expansion in C++
10. How to restrict dynamic allocation of objects
11. Sum of digits of a number in single statement
12. Write a URL in a C++ program
13. Zoom digits of an integer
14. Composite Design Pattern in C++
15. Assign value without any control statement

16. [Printing pyramid pattern](#)
17. [How to swap two variables in one line in C/C++, Python and Java?](#)
18. [Program to shut down a computer](#)

## Interview Questions

1. [Commonly Asked C++ Interview Questions | Set 1](#)
2. [Commonly Asked OOP Interview Questions | Set 1](#)
3. [C/C++ Programs](#)

## Quick Links:

- [Recent Articles on C++](#)
- [Practice Track on C++](#)
- [C++ Output & Multiple Choice Questions](#)

## Related Articles

1. Ruby Programming Language  
[<https://www.geeksforgeeks.org/ruby-programming-language/?ref=rp>]
2. Kotlin Programming Language  
[<https://www.geeksforgeeks.org/kotlin-programming-language/?ref=rp>]
3. Perl Programming Language  
[<https://www.geeksforgeeks.org/perl-programming-language/?ref=rp>]
4. Scala Programming Language  
[<https://www.geeksforgeeks.org/scala-programming-language/?ref=rp>]
5. Learning the art of Competitive Programming  
[<https://www.geeksforgeeks.org/learning-art-competitive-programming/?ref=rp>]
6. GATE and Programming Multiple Choice Questions with Solutions  
[<https://www.geeksforgeeks.org/gate-programming-multiple-choice-questions-solutions/?ref=rp>]
7. Quizzes on Programming Languages  
[<https://www.geeksforgeeks.org/quizzes-on-programming-languages-gq/?ref=rp>]
8. Articles on Programming Languages  
[<https://www.geeksforgeeks.org/articles-on-programming-languages-gq/?ref=rp>]

**SALE!****GeeksforGeeks Courses Upto 25% Off Enroll Now!****Save 25% on Courses**

DSA

Data Structures

Algorithms

Interview Preparation

Data Science

T

# Introduction to C++ Programming Language

Difficulty Level : Easy • Last Updated : 20 Feb, 2023

[Read](#)[Discuss](#)[Courses](#)[Practice](#)[Video](#)

**C++** is a general-purpose programming language that was developed as an enhancement of the C language to include object-oriented paradigm. It is an imperative and a **compiled** language.

1. C++ is a high-level, general-purpose programming language designed for system and application programming. It was developed by Bjarne Stroustrup at Bell Labs in 1983 as an extension of the C programming language. C++ is an object-oriented, multi-paradigm language that supports procedural, functional, and generic programming styles.
2. One of the key features of C++ is its ability to support low-level, system-level programming, making it suitable for developing operating systems, device drivers, and other system software. At the same time, C++ also provides a rich set of libraries and features for high-level application programming, making it a popular choice for developing desktop applications, video games, and other complex applications.
3. C++ has a large, active community of developers and users, and a wealth of resources and tools available for learning and using the language. Some of the key features of C++ include:
4. Object-Oriented Programming: C++ supports object-oriented programming, allowing developers to create classes and objects and to define methods and properties for these objects.
5. Templates: C++ templates allow developers to write generic code that can work with any data type, making it easier to write reusable and flexible code.
6. Standard Template Library (STL): The STL provides a wide range of containers and algorithms for working with data, making it easier to write efficient and effective code.
7. Exception Handling: C++ provides robust exception handling capabilities, making it easier to write code that can handle errors and unexpected situations.

Overall, C++ is a powerful and versatile programming language that is widely used for a range of applications and is well-suited for both low-level system programming and high-level application development.

**Here are some simple C++ code examples to help you understand the language:**

### 1. Hello World:

---

#### C++

```
#include <iostream>

int main() {
    std::cout << "Hello, World!" << std::endl;
    return 0;
}
```

#### Output

AD

Hello, World!



C++ is a middle-level language rendering it the advantage of programming low-level (drivers, kernels) and even higher-level applications (games, GUI, desktop apps etc.). The basic syntax and code structure of both C and C++ are the same.

Some of the **features & key-points** to note about the programming language are as follows:

- **Simple:** It is a simple language in the sense that programs can be broken down into logical units and parts, has a rich library support and a variety of data-types.
- **Machine Independent but Platform Dependent:** A C++ executable is not platform-independent (compiled programs on Linux won't run on Windows), however they are machine independent.
- **Mid-level language:** It is a mid-level language as we can do both systems-programming (drivers, kernels, networking etc.) and build large-scale user applications (Media Players, Photoshop, Game Engines etc.)
- **Rich library support:** Has a rich library support (Both standard ~ built-in data structures, algorithms etc.) as well 3rd party libraries (e.g. Boost libraries) for fast and rapid development.
- **Speed of execution:** C++ programs excel in execution speed. Since, it is a compiled language, and also hugely procedural. Newer languages have extra in-built default features such as garbage-collection, dynamic typing etc. which slow the execution of the program overall. Since there is no additional processing overhead like this in C++, it is blazing fast.
- **Pointer and direct Memory-Access:** C++ provides pointer support which aids users to directly manipulate storage address. This helps in doing low-level programming (where one might need to have explicit control on the storage of variables).
- **Object-Oriented:** One of the strongest points of the language which sets it apart from C. Object-Oriented support helps C++ to make maintainable and extensible programs. i.e.

Large-scale applications can be built. Procedural code becomes difficult to maintain as code-size grows.

- **Compiled Language:** C++ is a compiled language, contributing to its speed.

### **Applications of C++:**

C++ finds varied usage in applications such as:

- Operating Systems & Systems Programming. e.g. *Linux-based OS (Ubuntu etc.)*
- Browsers (*Chrome & Firefox*)
- Graphics & Game engines (*Photoshop, Blender, Unreal-Engine*)
- Database Engines (*MySQL, MongoDB, Redis etc.*)
- Cloud/Distributed Systems

### **Here are some key points to keep in mind while working with C++:**

1. Object-Oriented Programming: C++ is an object-oriented programming language, which means that it allows you to define classes and objects to model real-world entities and their behavior.
2. Strong Type System: C++ has a strong type system, which means that variables have a specific type and that type must be respected in all operations performed on that variable.
3. Low-level Access: C++ provides low-level access to system resources, which makes it a suitable choice for system programming and writing efficient code.
4. Standard Template Library (STL): The STL provides a large set of pre-written algorithms and data structures that can be used to simplify your code and make it more efficient.
5. Cross-platform Compatibility: C++ can be compiled and run on multiple platforms, including Windows, MacOS, and Linux, making it a versatile language for developing cross-platform applications.
6. Performance: C++ is a compiled language, which means that code is transformed into machine code before it is executed. This can result in faster execution times and improved performance compared to interpreted languages like Python.
7. Memory Management: C++ requires manual memory management, which can lead to errors if not done correctly. However, this also provides more control over the program's memory usage and can result in more efficient memory usage.
8. Syntax: C++ has a complex syntax that can be difficult to learn, especially for beginners. However, with practice and experience, it becomes easier to understand and work with.

These are some of the key points to keep in mind when working with C++. By understanding these concepts, you can make informed decisions and write effective code in this language.

## Advantages of C++:

1. Performance: C++ is a compiled language, which means that its code is compiled into machine-readable code, making it one of the fastest programming languages.
2. Object-Oriented Programming: C++ supports object-oriented programming, which makes it easier to write and maintain large, complex applications.
3. Standard Template Library (STL): The STL provides a wide range of algorithms and data structures for working with data, making it easier to write efficient and effective code.
4. Platform Independent: C++ is a platform-independent language, meaning that code written in C++ can be compiled and run on a wide range of platforms, from desktop computers to mobile devices.
5. Large Community: C++ has a large, active community of developers and users, providing a wealth of resources and support for learning and using the language.

## Disadvantages of C++:

1. Steep Learning Curve: C++ can be challenging to learn, especially for beginners, due to its complexity and the number of concepts that need to be understood.
2. Verbose Syntax: C++ has a verbose syntax, which can make code longer and more difficult to read and maintain.
3. Error-Prone: C++ provides low-level access to system resources, which can lead to subtle errors that are difficult to detect and fix.

## Reference Books:

1. "The C++ Programming Language" by Bjarne Stroustrup
2. "Effective C++: 55 Specific Ways to Improve Your Programs and Designs" by Scott Meyers
3. "C++ Primer Plus" by Stephen Prata
4. "C++ For Dummies" by Stephen R. Davis
5. "Data Structures and Algorithm Analysis in C++" by Mark Allen Weiss

### Some interesting facts about C++:

Here are some awesome facts about C++ that may interest you:

6. The name of C++ signifies the evolutionary nature of the changes from C. "++" is the C increment operator.
7. C++ is one of the predominant languages for the development of all kind of technical and commercial software.
8. C++ introduces Object-Oriented Programming, not present in C. Like other things, C++ supports the four primary features of OOP: encapsulation, polymorphism, abstraction, and inheritance.

9. C++ got the OOP features from Simula67 Programming language.
10. A function is a minimum requirement for a C++ program to run.(at least main() function)

Introduction to C++ | Sample Video for C++ Foundation Course | Geeks...



792

## Related Articles

1. C Programming Language Standard
2. Tips and Tricks for Competitive Programmers | Set 2 (Language to be used for Competitive Programming)
3. Why Java Language is Slower Than CPP for Competitive Programming?
4. Introduction to Parallel Programming with OpenMP in C++
5. Why C++ is partially Object Oriented Language?
6. Convert C/C++ code to assembly language
7. kbhit in C language
8. ToDo App in C Language
9. Fast I/O for Competitive Programming
10. getchar\_unlocked() – Faster Input in C/C++ For Competitive Programming

Next

Setting up C++ Development Environment

**SALE!**  
**GeeksforGeeks Courses Upto 25% Off Enroll Now!**



Save 25% on Courses   DSA   Data Structures   Algorithms   Interview Preparation   Data Science   T

# Setting up C++ Development Environment

Difficulty Level : Basic • Last Updated : 20 Mar, 2023

[Read](#)   [Discuss](#)   [Courses](#)   [Practice](#)   [Video](#)

C++ is a general-purpose programming language and is widely used nowadays for competitive programming. It has imperative, object-oriented, and generic programming features.

C++ runs on lots of platforms like Windows, Linux, Unix, Mac, etc. Before we start programming with C++. We will need an environment to be set up on our local computer to compile and run our C++ programs successfully. If you do not want to set up a local environment you can also use online IDEs for compiling your program.

## Using Online IDE

IDE stands for an integrated development environment. IDE is a software application that provides facilities to a computer programmer for developing software. There are many online IDEs available that you can use to compile and run your programs easily without setting up a local development environment. The

[ide.geeksforgeeks.org](https://ide.geeksforgeeks.org) is one such IDE provided by GeeksforGeeks.

You can click on the Run on IDE button to run the program.

AD

## C++

```
// Using online ide of C++
#include <iostream>
using namespace std;

int main()
{
    cout << "Learning C++ at GeekforGeeks";
    return 0;
}
```

### Output

Learning C++ at GeekforGeeks

Time Complexity: O(1)

Auxiliary Space: O(1)

## Setting up a Local Environment

For setting up a C++ Integrated Development Environment (IDE) on your local machine you need to install two important software:

1. C++ Compiler
2. Text Editor

### 1. C++ Compiler

Once you have installed the text editor and saved your program in a file with the '.cpp' extension, you will need a C++ compiler to compile this file. A compiler is a computer program that converts high-level language into machine-understandable low-level language. In other words, we can say that it converts the source code written in a programming language into another computer language that the computer understands. For compiling a C++ program we will need a C++ compiler that will convert the source code

written in C++ into machine codes. Below are the details about setting up compilers on different platforms.

### Installing GNU GCC on Linux

We will install the GNU GCC compiler on Linux. To install and work with the GCC compiler on your Linux machine, proceed according to the below steps:

**A.** You have to first run the below two commands from your Linux terminal window:

```
sudo apt-get update  
sudo apt-get install gcc  
sudo apt-get install g++
```

**B.** This command will install the GCC compiler on your system. You may also run the below command:

```
sudo apt-get install build-essential
```

This command will install all the libraries which are required to compile and run a C++ program.

**C.** After completing the above step, you should check whether the GCC compiler is installed in your system correctly or not. To do this you have to run the below-given command from the Linux terminal:

```
g++ --version
```

**D.** If you have completed the above two steps without any errors, then your Linux environment is set up and ready to be used to compile C++ programs. In further steps, we will learn how to compile and run a C++ program on Linux using the GCC compiler.

**E.** Write your program in a text file and save it with any file name and .CPP extension. We have written a program to display "Hello World" and saved it in a file with the filename "helloworld.cpp" on the desktop.

**F.** Now you have to open the Linux terminal and move to the directory where you have saved your file. Then you have to run the below command to compile your file:

```
g++ filename.cpp -o any-name
```

**G.** *filename.cpp* is the name of your source code file. In our case, the name is "helloworld.cpp" and *any-name* can be any name of your choice. This name will be assigned

to the executable file which is created by the compiler after compilation. In our case, we choose *any-name* to be "hello".

We will run the above command as:

```
g++ helloworld.cpp -o hello
```

**H.** After executing the above command, you will see a new file is created automatically in the same directory where you have saved the source file and the name of this file is the name you chose as *any-name*.

Now to run your program you have to run the below command:

```
./hello
```

**I.** This command will run your program in the terminal windows.

## 2. Text Editor

Text Editors are the type of programs used to edit or write texts. We will use text editors to type our C++ programs. The normal extension of a text file is (.txt) but a text file containing a C++ program should be saved with a '.cpp' or '.c' extension. Files ending with the extension '.CPP' and '.C' are called source code files and they are supposed to contain source code written in C++ programming language. These extension helps the compiler to identify that the file contains a C++ program.

Before beginning programming with C++, one must have a text editor installed to write programs. Follow the below instructions to install popular code editors like VS Code and Code::Block on different Operating Systems like windows, Mac OS, etc.

### 1. Code::Blocks Installation

There are lots of IDE available that you can use to work easily with the C++ programming language. One of the popular IDE is **Code::Blocks**.

- To download Code::Blocks, select the setup package based on your OS from this link - [Code::Blocks Setup Packages](#).
- Once you have downloaded the setup file of Code::Blocks from the given link open it and follow the instruction to install.
- After successfully installing Code::Blocks, go to *File* menu -> Select *New* and *create an Empty* file.
- Now write your C++ program in this empty file and save the file with a '.cpp' extension.
- After saving the file with the '.cpp' extension, go to the *Build* menu and choose the *Build and Run* option.

### 2. XCode Mac OS X Installation

If you are a Mac user, you can download Xcode as a code editor.

- To download Xcode you have to visit the apple website or you can search for it on the apple app store. You may follow the link – [Xcode for MacOS](#) to download Xcode. You will find all the necessary installation instructions there.
- After successfully installing Xcode, open the Xcode application.
- To create a new project. Go to File menu -> select New -> select Project. This will create a new project for you.
- Now in the next window, you have to choose a template for your project. To choose a C++ template choose the *Application* option which is under the *OS X* section on the left sidebar. Now choose *command-line tools* from available options and hit the *Next* button.
- On the next window provide all the necessary details like 'name of organization', 'Product Name, etc. But make sure to choose the language as C++. After filling in the details hit the next button to proceed to further steps.
- Choose the location where you want to save your project. After this choose the *main.cpp* file from the directory list on the left sidebar.
- Now after opening the *main.cpp* file, you will see a pre-written c++ program or template provided. You may change this program as per your requirement. To run your C++ program you have to go to the *Product* menu and choose the *Run* option from the dropdown.

Another very easy-to-use and most popular IDE nowadays, is **VSC ( Visual Studio Code )**, for both Windows and Mac OS.

### 3. Installing VS Code on Windows

Start with installing [Visual Studio Code](#) as per your windows. Open the downloaded file and click Run -> (Accept the agreement) Next -> Next -> Next -> (check all the options) -> Next -> Install->Finish.

Now you'll be able to see the Visual Studio Code icon on your desktop.

- Download the MinGW from [the link](#).
- After Install, "Continue". Check all the Packages (Right Click -> Mark for Installation). Now, Click on Installation (left corner) -> Apply Changes. (This may take time)
- Open This PC -> C Drive -> MinGW -> Bin. (Copy this path)
- Right, Click on "This PC" -> Properties -> Advanced System Setting -> Environment variables -> (Select PATH in System variables) -> Edit -> New -> Paste the path here and OK.
- Go to Visual Studio Code, and Install some useful extensions (from the right sidebar, last icon(probably))-
  - C/C++
  - Code Runner

- Now, Go to Setting -> Settings -> Search for Terminal -> Go to the end of this page -> Check [ Code-runner: Run In Terminal ]

Yay! You are good to go now. Open any folder, create new files and Save them with the extension ".cpp".

#### 4. Installing VS Code on Mac OS

First of all, Install Visual Studio Code for Mac OS using [this link](#). Then We'll install the compiler MinGW. For this, we first need to install Homebrew.

To install Homebrew, Open Terminal (cmd + space). Write Terminal and hit Enter. In cmd copy the given command

```
arch -x86_64 ruby -e "$(curl -fsSL  
https://raw.githubusercontent.com/Homebrew/install/HEAD/install)" <  
/dev/null 2> /dev/null
```

This will download and install HomeBrew on your Mac system. This process may take time.

Now We'll install the MinGW compiler on Mac OS. Paste the given command in the terminal and press Enter.

```
arch -x86_64 brew install MinGW-w64
```

- This is also time taking process so be patient!
- Go to Visual Studio Code, and Install some useful extensions (from the right sidebar, last icon(probably))-
  - C/C++
  - Code Runner
- Now, Go to Setting -> Settings -> Search for Terminal -> Go to the end of this page -> Check [ Code-runner: Run In Terminal ]

Yay! You are good to go now. Now open any folder, create new files, and Save them with the extension ".cpp".

C++ Programming Language Tutorial | Setting up C++ Development En...



This article is contributed by **Harsh Agarwal**. If you like GeeksforGeeks and would like to contribute, you can also write an article using [write.geeksforgeeks.org](https://write.geeksforgeeks.org). See your article appearing on the GeeksforGeeks main page and help other Geeks. Please write comments if you find anything incorrect, or if you want to share more information about the topic discussed above.

914

## Related Articles

1. [Setting up a C++ Competitive Programming Environment](#)

---
2. [Setting up Sublime Text for C++ Competitive Programming Environment](#)

---
3. [Different Ways to Setting Up Environment For C++ Programming in Mac](#)

---
4. [Installing MinGW Tools for C/C++ and Changing Environment Variable](#)

---
5. [Top 10 Programming Languages for Blockchain Development](#)

---
6. [Setting Up Sublime Text For Competitive Coding in C++14 on Ubuntu](#)

---
7. [Setting up Sublime Text For Competitive Programming \(C++\) Using Fast Olympic Coding Plugin](#)

---
8. [C++ Error - Does not name a type](#)

---
9. [Execution Policy of STL Algorithms in Modern C++](#)

---
10. [C++ Program To Print Pyramid Patterns](#)

[Previous](#)[Next](#)

## Article Contributed By :



GeeksforGeeks

## Vote for difficulty

Current difficulty : [Basic](#)

**SALE!**  
**GeeksforGeeks Courses Upto 25% Off Enroll Now!**



Save 25% on Courses DSA Data Structures Algorithms Interview Preparation Data Science T

# C++ Programming Basics

Difficulty Level : Easy • Last Updated : 11 Mar, 2023

[Read](#) [Discuss](#) [Courses](#) [Practice](#) [Video](#)

C++ is a general-purpose programming language and is widely used nowadays for competitive programming. It has imperative, object-oriented, and generic programming features. C++ runs on lots of platforms like Windows, Linux, Unix, Mac, etc.

Before explaining the basics of C++, we would like to clarify two more ideas: **low-level** and **high-level**. To make it easy to understand, let's consider this scenario – when we go to the Google search engine and search for some queries, Google displays some websites according to our question. Google does this for us at a very high level. We don't know what's happening at the low level until we look into Google servers (at a low level) and further to the level where the data is in the form of 0s/1s. The point we want to make here is that a low level means nearest to the hardware, and a high level means farther from the hardware with a lot of layers of abstraction. **C ++ is considered a low-level language** as it is closer to hardware than most general-purpose programming languages. However to become proficient in any programming language, one Firstly needs to understand the basics of that language.

## Basics of C++ Programming

### 1. Basic Syntax and First Program in C++

Learning C++ programming can be simplified into writing your program in a text editor and saving it with the correct extension (.CPP, C, CP), and compiling your program using a compiler or online IDE. The "Hello World" program is the first step toward learning any programming language and is also one of the simplest programs you will learn.

We can learn more about C++ Basic Syntax here – [C++ Basic Syntax](#)

AD

## 2. Basic Input and Output in C++

C++ comes with libraries that provide us with many ways for performing input and output. In C++ input and output are performed in the form of a sequence of bytes or more commonly known as streams. The two methods **cin** and **cout** are used very often for taking inputs and printing outputs respectively. These two are the most basic methods of taking input and output in C++.

To learn more about basic input and output in C++, refer to this article – [Basic Input/Output in C++](#)

## 3. Comments in C++

A well-documented program is a good practice for a programmer. It makes a program more readable and error finding becomes easier. One important part of good documentation is Comments. In computer programming, a comment is a programmer-readable explanation or annotation in the source code of a computer program. These are statements that are not executed by the compiler and interpreter.

We can learn more about C++ comments in this article – [C++ Comments](#)

## 4. Data Types and Modifiers in C++

All variables use data type during declaration to restrict the type of data to be stored. Therefore, we can say that data types are used to tell the variables the type of data they can store. Whenever a variable is defined in C++, the compiler allocates some memory for that variable based on the data type with which it is declared. Every data type requires a different amount of memory. We can change this by using data type modifiers.

To know more about data types, refer to this article – [C++ Data Types](#)

To know more about type modifiers, refer to this article – [C++ Type Modifiers](#)

## 5. Variables in C++

A variable is a name given to a memory location. It is the basic unit of storage in a program. The value stored in a variable can be changed during program execution. A variable is only a name given to a memory location, all the operations done on the variable effects that memory location. In C++, all the variables must be declared before use.

To know more about C++ variables, refer to this article – [C++ Variables](#)

## 6. Variable Scope in C++

In general, the scope is defined as the extent to which something can be worked with. In programming also the scope of a variable is defined as the extent of the program code within which the variable can be accessed or declared or worked with. There are mainly two types of variable scopes, Local and Global Variables.

To know more about variable scope in C++, refer to this article – [Scope of Variable in C++](#)

## 7. Uninitialized Variable in C++

“One of the things that have kept C++ viable is the zero-overhead rule: What you don’t use, you don’t pay for.” -Stroustrup. The overhead of initializing a stack variable is costly as it hampers the speed of execution, therefore these variables can contain indeterminate values. It is considered a best practice to initialize a primitive data type variable before using it in code.

To know more about what happens to the uninitialized variables, refer to this article – [Uninitialized primitive data types in C/C++](#)

## 8. Constants and Literals in C++

As the name suggests the name constants are given to such variables or values in C++ programming language which cannot be modified once they are defined. They are fixed values in a program. There can be any type of constant like integer, float, octal, hexadecimal, character constants, etc. Every constant has some range. The integers that are too big to fit into an int will be taken as long. Now there are various ranges that differ from unsigned to signed bits. Under the signed bit, the range of an int varies from -128 to +127, and under the unsigned bit, the int varies from 0 to 255. Literals are a kind of constant and both terms are used interchangeably in C++.

To know more about constants in C++, refer to this article – [Constants in C++](#)

To know more about literals in C++, refer to this article – [Literals in C++](#)

## 9. Operators in C++

Operators are the foundation of any programming language. Thus the functionality of the C/C++ programming language is incomplete without the use of operators. We can define operators as symbols that help us to perform specific mathematical and logical computations on operands. In other words, we can say that an operator operates the operands.

To know more about C++ operators, refer to this article - [Operators in C++](#)

## 10. Loops in C++

Loops in programming come into use when we need to repeatedly execute a block of statements. For example: Suppose we want to print "Hello World" 10 times. This can be done in two ways, the Iterative method and using Loops.

To know more about loops in C++, refer to this article - [C++ Loops](#)

## 11. Decision-Making in C++

There comes situations in real life when we need to make some decisions and based on these decisions, we decide what should we do next. Similar situations arise in programming also where we need to make some decisions and based on these decisions we will execute the next block of code. Decision-making statements in programming languages decide the direction of the flow of program execution.

To know more about decision-making statements in C++, refer to this article - [Decision Making in C++](#)

## 12. Classes and Objects in C++

Classes and Objects are used to provide Object Oriented Programming in C++. A class is a user-defined datatype that is a blueprint of an object that contains data members (attribute) and member methods that works on that data. Objects are the instances of a class that are required to use class methods and data as we cannot use them directly.

To know more about classes and objects, refer to this article - [C++ Classes and Objects](#)

## 13. Access Modifiers in C++

Access modifiers are used to implement an important feature of Object-Oriented Programming known as Data Hiding. Access modifiers or Access Specifiers in a class are

used to set the accessibility of the class members. That is, it sets some restrictions on the class members not to get directly accessed by outside functions.

To know more about access modifiers in C++, refer to this article – [Access Modifiers in C++](#)

## 14. Storage Classes in C++

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

To know more about storage classes in C++, refer to this article – [Storage Classes in C++](#)

## 15. Forward declarations in C++

It refers to the beforehand declaration of the syntax or signature of an identifier, variable, function, class, etc. prior to its usage (done later in the program). In C++, Forward declarations are usually used for Classes. In this, the class is pre-defined before its use so that it can be called and used by other classes that are defined before this.

To know more about forward declarations in C++, refer to this article – [What are forward declarations in C++?](#)

## 16. Errors in C++

An error is an illegal operation performed by the user which results in the abnormal working of the program. Programming errors often remain undetected until the program is compiled or executed. Some of the errors inhibit the program from getting compiled or executed. Thus errors should be removed before compiling and executing.

To know more about C++ errors, refer to this article – [Errors in C++](#)

## 17. Undefined Behaviour in C++

If a user starts learning in a C/C++ environment and is unclear about the concept of undefined behavior then that can bring plenty of problems in the future while debugging someone else code might be actually difficult in tracing the root to the undefined error.

To know more about undefined behavior, refer to this article – [Undefined Behavior in C and C++](#)

**SALE!**  
**GeeksforGeeks Courses Upto 25% Off Enroll Now!**



Save 25% on Courses   DSA   Data Structures   Algorithms   Array   Strings   Linked List   Stack   Q

# C++ Data Types

Difficulty Level : Basic • Last Updated : 18 Mar, 2023

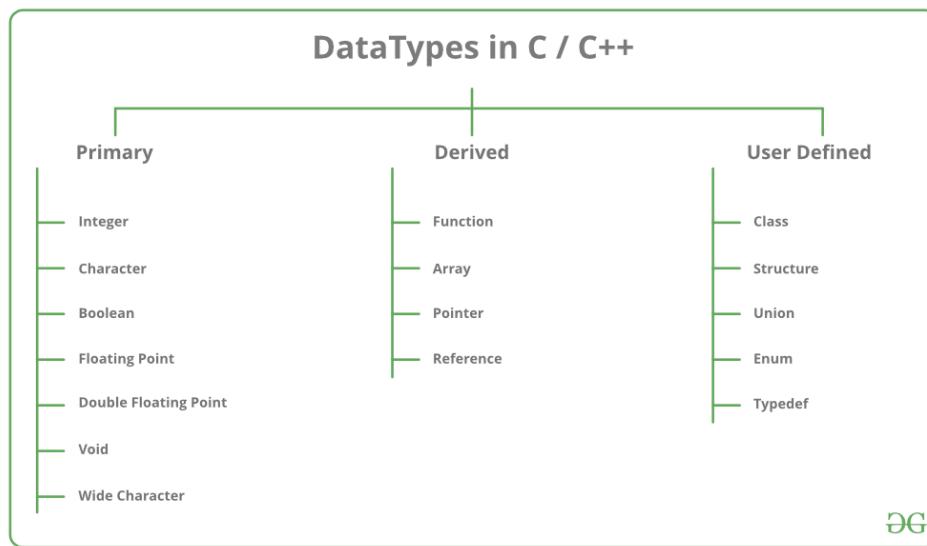
[Read](#)   [Discuss\(30+\)](#)   [Courses](#)   [Practice](#)   [Video](#)

All variables use data type during declaration to restrict the type of data to be stored. Therefore, we can say that data types are used to tell the variables the type of data they can store. Whenever a variable is defined in C++, the compiler allocates some memory for that variable based on the data type with which it is declared. Every data type requires a different amount of memory.

C++ supports a wide variety of data types and the programmer can select the data type appropriate to the needs of the application. Data types specify the size and types of values to be stored. However, storage representation and machine instructions to manipulate each data type differ from machine to machine, although C++ instructions are identical on all machines.

## C++ supports the following data types:

1. **Primary or Built-in or Fundamental data type**
2. **Derived data types**
3. **User-defined data types**



AD

## Data Types in C++ are Mainly Divided into 3 Types:

**1. Primitive Data Types:** These data types are built-in or predefined data types and can be used directly by the user to declare variables. example: int, char, float, bool, etc. Primitive data types available in C++ are:

- Integer
- Character
- Boolean
- Floating Point
- Double Floating Point
- Valueless or Void
- Wide Character

**2. Derived Data Types:** [Derived data types](#) that are derived from the primitive or built-in datatypes are referred to as Derived Data Types. These can be of four types namely:

- Function
- Array
- Pointer
- Reference

**3. Abstract or User-Defined Data Types:** [Abstract or User-Defined data types](#) are defined by the user itself. Like, defining a class in C++ or a structure. C++ provides the following user-defined datatypes:

- Class
- Structure
- Union
- Enumeration
- Typedef defined Datatype

## Primitive Data Types

- **Integer:** The keyword used for integer data types is **int**. Integers typically require 4 bytes of memory space and range from -2147483648 to 2147483647.
- **Character:** Character data type is used for storing characters. The keyword used for the character data type is **char**. Characters typically require 1 byte of memory space and range from -128 to 127 or 0 to 255.
- **Boolean:** Boolean data type is used for storing Boolean or logical values. A Boolean variable can store either *true* or *false*. The keyword used for the Boolean data type is **bool**.
- **Floating Point:** Floating Point data type is used for storing single-precision floating-point values or decimal values. The keyword used for the floating-point data type is **float**. Float variables typically require 4 bytes of memory space.
- **Double Floating Point:** Double Floating Point data type is used for storing double-precision floating-point values or decimal values. The keyword used for the double floating-point data type is **double**. Double variables typically require 8 bytes of memory space.
- **void:** Void means without any value. void data type represents a valueless entity. A void data type is used for those function which does not return a value.
- **Wide Character:** [Wide character](#) data type is also a character data type but this data type has a size greater than the normal 8-bit data type. Represented by **wchar\_t**. It is generally 2 or 4 bytes long.
- **sizeof() operator:** [sizeof\(\) operator](#) is used to find the number of bytes occupied by a variable/data type in computer memory.

### Example:

```
int m, x[50];
```

```
cout<<sizeof(m); //returns 4 which is the number of bytes occupied by the integer variable "m".
```

`cout<<sizeof(x); //returns 200 which is the number of bytes occupied by the integer array variable "x".`

The size of variables might be different from those shown in the above table, depending on the compiler and the computer you are using.

---

## C++

```
// C++ Program to Demonstrate the correct size  
// of various data types on your computer.  
#include <iostream>  
using namespace std;  
  
int main()  
{  
    cout << "Size of char : " << sizeof(char) << endl;  
    cout << "Size of int : " << sizeof(int) << endl;  
  
    cout << "Size of long : " << sizeof(long) << endl;  
    cout << "Size of float : " << sizeof(float) << endl;  
  
    cout << "Size of double : " << sizeof(double) << endl;  
  
    return 0;  
}
```

## Output

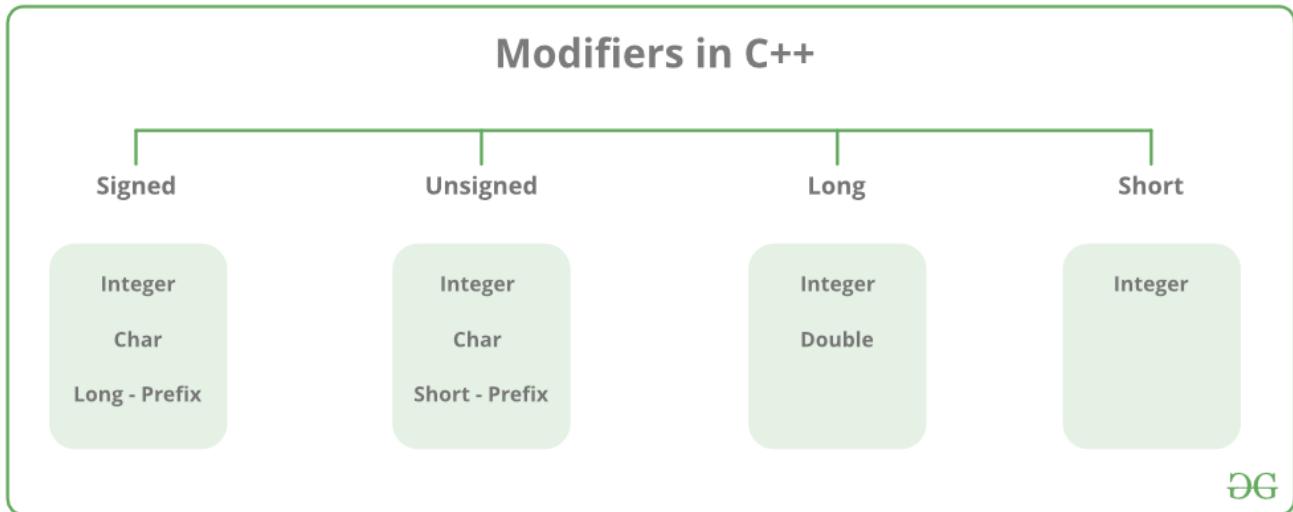
```
Size of char : 1  
Size of int : 4  
Size of long : 8  
Size of float : 4  
Size of double : 8
```

**Time Complexity:** O(1)

**Space Complexity:** O(1)

## Datatype Modifiers

As the name suggests, datatype modifiers are used with built-in data types to modify the length of data that a particular data type can hold.



Data type modifiers available in C++ are:

- **Signed**
- **Unsigned**
- **Short**
- **Long**

The below table summarizes the modified size and range of built-in datatypes when combined with the type modifiers:

Data Type	Size (in bytes)	Range
short int	2	-32,768 to 32,767
unsigned short int	2	0 to 65,535
unsigned int	4	0 to 4,294,967,295
int	4	-2,147,483,648 to 2,147,483,647
long int	4	-2,147,483,648 to 2,147,483,647
unsigned long int	4	0 to 4,294,967,295
long long int	8	-(2^63) to (2^63)-1

Data Type	Size (in bytes)	Range
unsigned long long int	8	0 to 18,446,744,073,709,551,615
signed char	1	-128 to 127
unsigned char	1	0 to 255
float	4	$-3.4 \times 10^{38}$ to $3.4 \times 10^{38}$
double	8	$-1.7 \times 10^{308}$ to $1.7 \times 10^{308}$
long double	12	$-1.1 \times 10^{4932}$ to $1.1 \times 10^{4932}$
wchar_t	2 or 4	1 wide character

**Note:** Above values may vary from compiler to compiler. In the above example, we have considered GCC 32 bit.

We can display the size of all the data types by using the `sizeof()` operator and passing the keyword of the datatype, as an argument to this function as shown below:

Now to get the range of data types refer to the following chart

**Note:** `syntax<limits.h>` header file is defined to find the range of fundamental data-types. Unsigned modifiers have minimum value is zero. So, no macro constants are defined for the unsigned minimum value.

## Macro Constants

Name	Expresses
CHAR_MIN	The minimum value for an object of type char
CHAR_MAX	Maximum value for an object of type char
SCHAR_MIN	The minimum value for an object of type Signed char
SCHAR_MAX	Maximum value for an object of type Signed char
UCHAR_MAX	Maximum value for an object of type Unsigned char
CHAR_BIT	Number of bits in a char object
MB_LEN_MAX	Maximum number of bytes in a multi-byte character
SHRT_MIN	The minimum value for an object of type short int
SHRT_MAX	Maximum value for an object of type short int
USHRT_MAX	Maximum value for an object of type Unsigned short int
INT_MIN	The minimum value for an object of type int
INT_MAX	Maximum value for an object of type int
UINT_MAX	Maximum value for an object of type Unsigned int
LONG_MIN	The minimum value for an object of type long int
LONG_MAX	Maximum value for an object of type long int

Name	Expresses
ULONG_MAX	Maximum value for an object of type Unsigned long int
LLONG_MIN	The minimum value for an object of type long long int
LLONG_MAX	Maximum value for an object of type long long int
ULLONG_MAX	Maximum value for an object of type Unsigned long long int

The actual value depends on the particular system and library implementation but shall reflect the limits of these types in the target platform. LLONG\_MIN, LLONG\_MAX, and ULLONG\_MAX are defined for libraries complying with the C standard of 1999 or later (which only includes the C++ standard since 2011: C++11).

## C++ Program to Find the Range of Data Types using Macro Constants

### Example:

### C++

```
// C++ program to Demonstrate the sizes of data types
#include <iostream>
#include <limits.h>
using namespace std;

int main()
{
    cout << "Size of char : " << sizeof(char) << " byte"
        << endl;

    cout << "char minimum value: " << CHAR_MIN << endl;

    cout << "char maximum value: " << CHAR_MAX << endl;

    cout << "Size of int : " << sizeof(int) << " bytes"
        << endl;

    cout << "Size of short int : " << sizeof(short int)
        << " bytes" << endl;

    cout << "Size of long int : " << sizeof(long int)
        << " bytes" << endl;

    cout << "Size of signed long int : "
        << sizeof(signed long int) << " bytes" << endl;
```

```

cout << "Size of unsigned long int : "
    << sizeof(unsigned long int) << " bytes" << endl;

cout << "Size of float : " << sizeof(float) << " bytes"
    << endl;

cout << "Size of double : " << sizeof(double)
    << " bytes" << endl;

cout << "Size of wchar_t : " << sizeof(wchar_t)
    << " bytes" << endl;

return 0;
}

```

## Output

```

Size of char : 1 byte
char minimum value: -128
char maximum value: 127
Size of int : 4 bytes
Size of short int : 2 bytes
Size of long int : 8 bytes
Size of signed long int : 8 bytes
Size of unsigned long int : 8 bytes
Size of float : 4 bytes
Size of double : 8 bytes
Size of wchar_t : 4 bytes

```

**Time Complexity:** O(1)

**Space Complexity:** O(1)

---

## C++

```

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

int main() {
    // Integer data types
    int a = 10;
    short b = 20;
    long c = 30;
    long long d = 40;
    cout << "Integer data types: " << endl;
    cout << "int: " << a << endl;
    cout << "short: " << b << endl;
    cout << "long: " << c << endl;
    cout << "long long: " << d << endl;
}

```

```
// Floating-point data types
float e = 3.14f;
double f = 3.141592;
long double g = 3.14159265358979L;
cout << "Floating-point data types: " << endl;
cout << "float: " << e << endl;
cout << "double: " << f << endl;
cout << "long double: " << g << endl;

// Character data types
char h = 'a';
wchar_t i = L'b';
char16_t j = u'c';
char32_t k = U'd';
cout << "Character data types: " << endl;
cout << "char: " << h << endl;
wcout << "wchar_t: " << i << endl;
cout << "char16_t: " << j << endl;
cout << "char32_t: " << k << endl;

// Boolean data type
bool l = true;
bool m = false;
cout << "Boolean data type: " << endl;
cout << "true: " << l << endl;
cout << "false: " << m << endl;

// String data type
string n = "Hello, world!";
cout << "String data type: " << endl;
cout << n << endl;

return 0;
}
```

## Output

Integer data types:

int: 10  
short: 20  
long: 30  
long long: 40

Floating-point data types:

float: 3.14  
double: 3.14159  
long double: 3.14159

Character data types:

char: a  
wchar\_t: b  
char16\_t: 99

```
char32_t: 100
Boolean data type:
true: 1
false: 0
String data type:
Hello, world!
```

This program declares variables of various data types, assigns values to them, and then prints out their values.

The integer data types include int, short, long, and long long. These data types represent whole numbers of varying sizes.

The floating-point data types include float, double, and long double. These data types represent real numbers with varying levels of precision.

The character data types include char, wchar\_t, char16\_t, and char32\_t. These data types represent individual characters of varying sizes.

The boolean data type is a simple data type that can only have one of two values: true or false.

The string data type is a sequence of characters. In this program, we use the string class to declare a string variable and assign it a value.

## **Advantages:**

Data types provide a way to categorize and organize data in a program, making it easier to understand and manage.

Each data type has a specific range of values it can hold, allowing for more precise control over the type of data being stored.

Data types help prevent errors and bugs in a program by enforcing strict rules about how data can be used and manipulated.

C++ provides a wide range of data types, allowing developers to choose the best type for a specific task.

## **Disadvantages:**

Using the wrong data type can result in unexpected behavior and errors in a program.

Some data types, such as long doubles or char arrays, can take up a large amount of memory and impact performance if used excessively.

C++'s complex type system can make it difficult for beginners to learn and use the language effectively.

The use of data types can add additional complexity and verbosity to a program, making it harder to read and understand.

This article is contributed by **Harsh Agarwal**. If you like GeeksforGeeks and would like to contribute, you can also write an article using [write.geeksforgeeks.org](https://write.geeksforgeeks.org) or mail your article to review-team@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks. Please write comments if you find anything incorrect, or if you want to share more information about the topic discussed above.

1.12k

## Related Articles

1. Difference between fundamental data types and derived data types
2. What Happen When We Exceed Valid Range of Built-in Data Types in C++?
3. Calculate range of data types using C++
4. Uninitialized primitive data types in C/C++
5. User defined Data Types in C++
6. Different types of Coding Schemes to represent data
7. Derived Data Types in C++
8. Data types that supports std::numeric\_limits() in C++
9. Data Communication - Definition, Components, Types, Channels
10. C++ Char Data Types

[Previous](#)[Next](#)

## Article Contributed By :



## Vote for difficulty

Current difficulty : [Basic](#)

SALE!  
GeeksforGeeks Courses Upto 25% Off Enroll Now!



Save 25% on Courses DSA Data Structures Algorithms Interview Preparation Data Science T

# C++ Variables

Difficulty Level : Easy • Last Updated : 16 Mar, 2023

[Read](#) [Discuss](#) [Courses](#) [Practice](#) [Video](#)

Variables in C++ is a name given to a memory location. It is the basic unit of storage in a program.

- The value stored in a variable can be changed during program execution.
- A variable is only a name given to a memory location, all the operations done on the variable effects that memory location.
- In C++, all the variables must be declared before use.

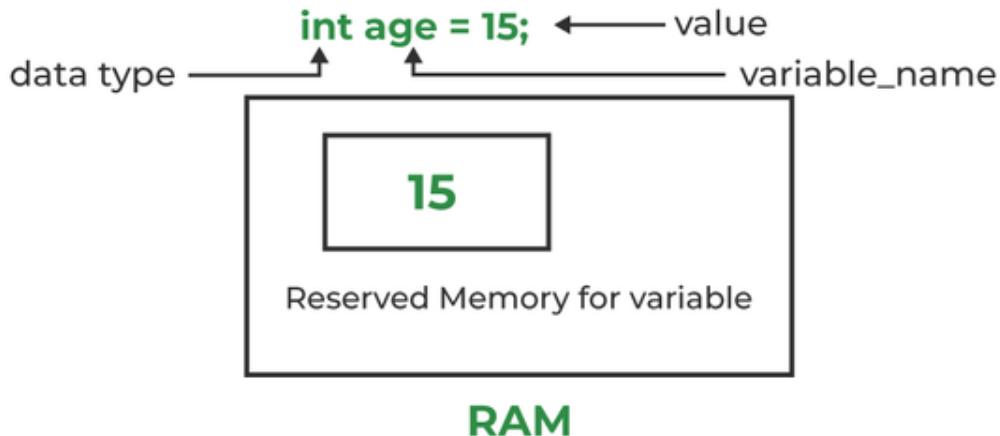
## How to Declare Variables?

A typical variable declaration is of the form:

```
// Declaring a single variable  
type variable_name;  
  
// Declaring multiple variables:  
type variable1_name, variable2_name, variable3_name;
```

A variable name can consist of alphabets (both upper and lower case), numbers, and the underscore '\_' character. However, the name must not start with a number.

## Variable in C++



*Initialization of a variable in C++*

In the above diagram,

AD

**datatype:** Type of data that can be stored in this variable.

**variable\_name:** Name given to the variable.

**value:** It is the initial value stored in the variable.

**Examples:**

```
// Declaring float variable  
float simpleInterest;
```

```
// Declaring integer variable  
int time, speed;
```

```
// Declaring character variable  
char var;
```

We can also provide values while declaring the variables as given below:

```
int a=50,b=100; //declaring 2 variable of integer type
float f=50.8; //declaring 1 variable of float type
char c='Z'; //declaring 1 variable of char type
```

## Rules For Declaring Variable

- The name of the variable contains letters, digits, and underscores.
- The name of the variable is case sensitive (ex Arr and arr both are different variables).
- The name of the variable does not contain any whitespace and special characters (ex #,\$,%,\* , etc).
- All the variable names must begin with a letter of the alphabet or an underscore(\_).
- We cannot used C++ keyword(ex float,double,class)as a variable name.

### Valid variable names:

```
int x; //can be letters
int _yz; //can be underscores
int z40;//can be letters
```

### Invalid variable names:

```
int 89; Should not be a number
int a b; //Should not contain any whitespace
int double;// C++ keyword CAN NOT BE USED
```

## Difference Between Variable Declaration and Definition

The **variable declaration** refers to the part where a variable is first declared or introduced before its first use. A **variable definition** is a part where the variable is assigned a memory location and a value. Most of the time, variable declaration and definition are done together.

See the following C++ program for better clarification:

### C++

```
// C++ program to show difference between
// definition and declaration of a
// variable
#include <iostream>
using namespace std;

int main()
{
    // this is declaration of variable a
```

```
int a;

// this is initialisation of a
a = 10;

// this is definition = declaration + initialisation
int b = 20;

// declaration and definition
// of variable 'a123'
char a123 = 'a';

// This is also both declaration and definition
// as 'c' is allocated memory and
// assigned some garbage value.
float c;

// multiple declarations and definitions
int _c, _d45, e;

// Let us print a variable
cout << a123 << endl;

return 0;
}
```

## Output

a

**Time Complexity:** O(1)

**Space Complexity:** O(1)

## Types of Variables

There are three types of variables based on the scope of variables in C++

- **Local Variables**
- **Instance Variables**
- **Static Variables**

## Type of variables in C++

```
class GFG {
    public :
        static int a ; → Static Variable
        int b ; → Instance Variable
    public :
        func ()
        {
            int c ; → Local Variable
        },
};
```

*Types of Variables in C++*

Let us now learn about each one of these variables in detail.

**1. Local Variables:** A variable defined within a block or method or constructor is called a local variable.

- These variables are created when entered into the block or the function is called and destroyed after exiting from the block or when the call returns from the function.
- The scope of these variables exists only within the block in which the variable is declared. i.e. we can access this variable only within that block.
- Initialization of Local Variable is Mandatory.

**2. Instance Variables:** Instance variables are non-static variables and are declared in a class outside any method, constructor, or block.

- As instance variables are declared in a class, these variables are created when an object of the class is created and destroyed when the object is destroyed.
- Unlike local variables, we may use access specifiers for instance variables. If we do not specify any access specifier then the default access specifier will be used.
- Initialization of Instance Variable is not Mandatory.
- Instance Variable can be accessed only by creating objects.

**3. Static Variables:** Static variables are also known as Class variables.

- These variables are declared similarly as instance variables, the difference is that static variables are declared using the static keyword within a class outside any method constructor or block.
- Unlike instance variables, we can only have one copy of a static variable per class irrespective of how many objects we create.
- Static variables are created at the start of program execution and destroyed automatically when execution ends.

- Initialization of Static Variable is not Mandatory. Its default value is 0
- If we access the static variable like the Instance variable (through an object), the compiler will show the warning message and it won't halt the program. The compiler will replace the object name with the class name automatically.
- If we access the static variable without the class name, the Compiler will automatically append the class name.

## Instance Variable Vs Static Variable

- Each object will have its **own copy** of the instance variable whereas We can only have **one copy** of a static variable per class irrespective of how many objects we create.
- Changes made in an instance variable using one object will **not be reflected** in other objects as each object has its own copy of the instance variable. In the case of static, changes **will be reflected** in other objects as static variables are common to all objects of a class.
- We can access instance variables **through object references** and Static Variables can be accessed **directly using the class name**.
- The syntax for static and instance variables:

```
class Example
{
    static int a; // static variable
    int b;        // instance variable
}
```

384

## Related Articles

1. Templates and Static variables in C++
2. Swap Two Variables in One Line
3. Scope of Variables in C++
4. Can Global Variables be dangerous ?
5. C++ 17 | New ways to Assign values to Variables
6. Why do we need reference variables if we have pointers

**SALE!**  
**GeeksforGeeks Courses Upto 25% Off Enroll Now!**



Save 25% on Courses DSA Data Structures Algorithms Interview Preparation Data Science T

# C++ Loops

Difficulty Level : Easy • Last Updated : 18 Mar, 2023

Read Discuss Courses Practice Video

In Programming, sometimes there is a need to perform some operation **more than once** or (say) **n number** of times. Loops come into use when we need to repeatedly execute a block of statements.

**For example:** Suppose we want to print "Hello World" 10 times. This can be done in two ways as shown below:

## Manual Method (Iterative Method)

Manually we have to write **cout** for the C++ statement 10 times. Let's say you have to write it 20 times (it would surely take more time to write 20 statements) now imagine you have to write it 100 times, it would be really hectic to re-write the same statement again and again. So, here loops have their role.

## C++

```
// C++ program to Demonstrate the need of loops
#include <iostream>
using namespace std;

int main()
{
    cout << "Hello World\n";
    return 0;
}
```

## Output

AD

```
Hello World  
Hello World  
Hello World  
Hello World  
Hello World
```

**Time complexity:** O(1)

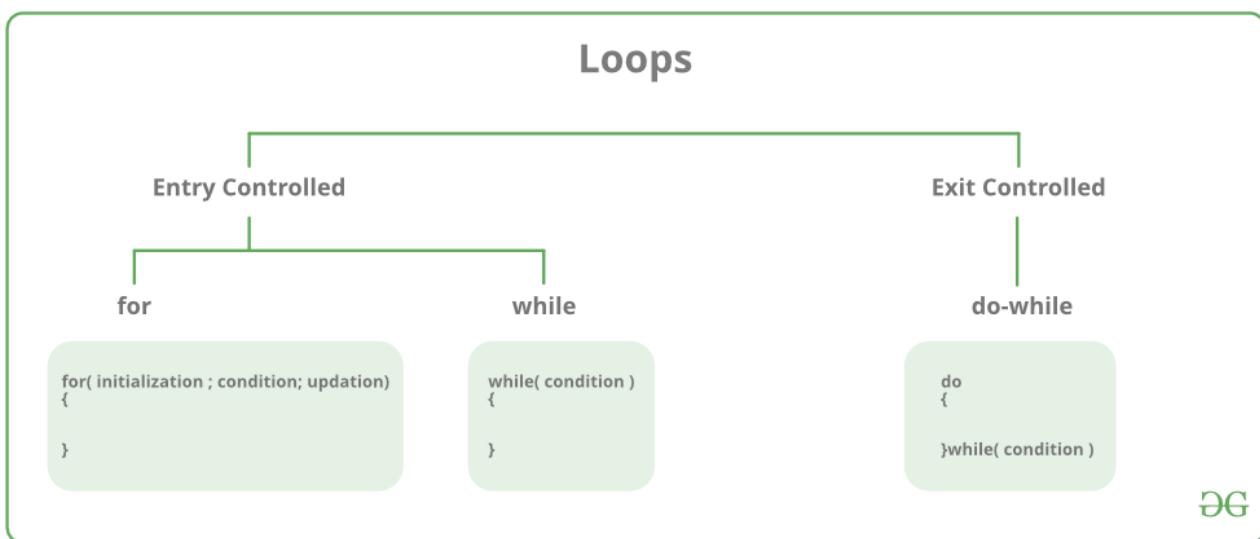
**Space complexity:** O(1)

## Using Loops

In Loop, the statement needs to be written only once and the loop will be executed 10 times as shown below. In computer programming, a loop is a sequence of instructions that is repeated until a certain condition is reached.

### There are mainly two types of loops:

1. **Entry Controlled loops:** In this type of loop, the test condition is tested before entering the loop body. **For Loop** and **While Loop** is entry-controlled loops.
2. **Exit Controlled Loops:** In this type of loop the test condition is tested or evaluated at the end of the loop body. Therefore, the loop body will execute at least once, irrespective of whether the test condition is true or false. the do-while **loop** is exit controlled loop.



S.No.	Loop Type and Description
1.	<b>while loop</b> – First checks the condition, then executes the body.
2.	<b>for loop</b> – firstly initializes, then, condition check, execute body, update.
3.	<b>do-while loop</b> – firstly, execute the body then condition check

## For Loop-

A *For loop* is a repetition control structure that allows us to write a loop that is executed a specific number of times. The loop enables us to perform n number of steps together in one line.

## Syntax:

```
for (initialization expr; test expr; update expr)
{
    // body of the loop
    // statements we want to execute
}
```

## **Explanation of the Syntax:**

- **Initialization statement:** This statement gets executed only once, at the beginning of the for loop. You can enter a declaration of multiple variables of one type, such as int x=0, a=1, b=2. These variables are only valid in the scope of the loop. Variable defined before the loop with the same name are hidden during execution of the loop.

- **Condition:** This statement gets evaluated ahead of each execution of the loop body, and abort the execution if the given condition get false.
- **Iteration execution:** This statement gets executed after the loop body, ahead of the next condition evaluated, unless the for loop is aborted in the body (by break, goto, return or an exception being thrown.)

## NOTES:

- The initialization and increment statements can perform operations unrelated to the condition statement, or nothing at all – if you wish to do. But the good practice is to only perform operations directly relevant to the loop.
- A variable declared in the initialization statement is visible only inside the scope of the for loop and will be released out of the loop.
- Don't forget that the variable which was declared in the initialization statement can be modified during the loop, as well as the variable checked in the condition.

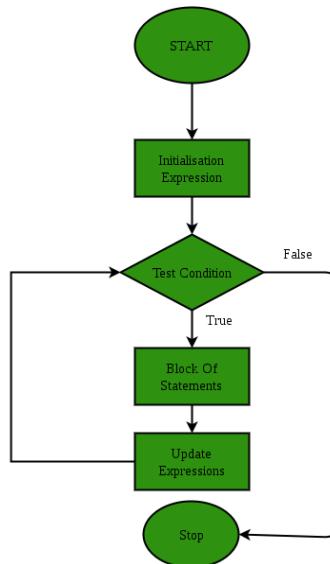
### **Example1:**

```
for(int i = 0; i < n; i++)
{
    // BODY
}
```

### **Example2:**

```
for(auto element:arr)
{
    //BODY
}
```

### **Flow Diagram of for loop:**

**Example1:****C++**

```
// C++ program to Demonstrate for loop
#include <iostream>
using namespace std;

int main()
{
    for (int i = 1; i <= 5; i++) {
        cout << "Hello World\n";
    }

    return 0;
}
```

**Output**

```
Hello World
Hello World
Hello World
Hello World
Hello World
```

**Time complexity:** O(1)**Space complexity:** O(1)**Example2:****C++**

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

int main() {

int arr[] {40, 50, 60, 70, 80, 90, 100};
for (auto element: arr){
    cout << element << " ";
}
return 0;

}
```

## Output

40 50 60 70 80 90 100

**Time complexity:** O(n) n is the size of array.

**Space complexity:** O(n) n is the size of array.

**For loop can also be valid in the given form:-**

---

## C++

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

int main()
{
    for (int i = 0, j = 10, k = 20; (i + j + k) < 100;
         j++, k--, i += k) {
        cout << i << " " << j << " " << k << "\n";
    }
    return 0;
}
```

## Output

0 10 20  
19 11 19  
37 12 18  
54 13 17

**Time complexity:** O(1)

**Space complexity:** O(1)

**Example of hiding declared variables before a loop is:**

---

## C++

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

int main()
{
    int i = 99;
    for (int i = 0; i < 5; i++) {
        cout << i << "\t";
    }
    cout << "\n" << i;
    return 0;
}
```

### Output

0     1     2     3     4  
99

**Time complexity:** O(1)

**Space complexity:** O(1)

But if you want to use the already declared variable and not hide it, then must not redeclare that variable.

### For Example:

## C++

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

int main()
{
    int i = 99;
    for (i = 0; i < 5; i++) {
        cout << i << " ";
    }
    cout << "\n" << i;
    return 0;
}
```

### Output

0 1 2 3 4  
5

**Time complexity:** O(1)**Space complexity:** O(1)

The For loop can be used to iterating through the elements in the STL container(e.g., Vector, etc). here we have to use iterator.

**For Example:****C++**

```
#include <bits/stdc++.h>
using namespace std;

int main()
{
    vector<int> v = { 1, 2, 3, 4, 5 };
    for (vector<int>::iterator it = v.begin();
         it != v.end(); it++) {
        cout << *it << "\t";
    }
    return 0;
}
```

**Output**

1    2    3    4    5

**Time complexity:** O(n) n is the size of vector.**Space complexity:** O(n) n is the size of vector.**While Loop-**

While studying **for loop** we have seen that the number of iterations is ***known beforehand***, i.e. the number of times the loop body is needed to be executed is known to us. while loops are used in situations where **we do not know** the exact number of iterations of the loop ***beforehand***. The loop execution is terminated on the basis of the test conditions.

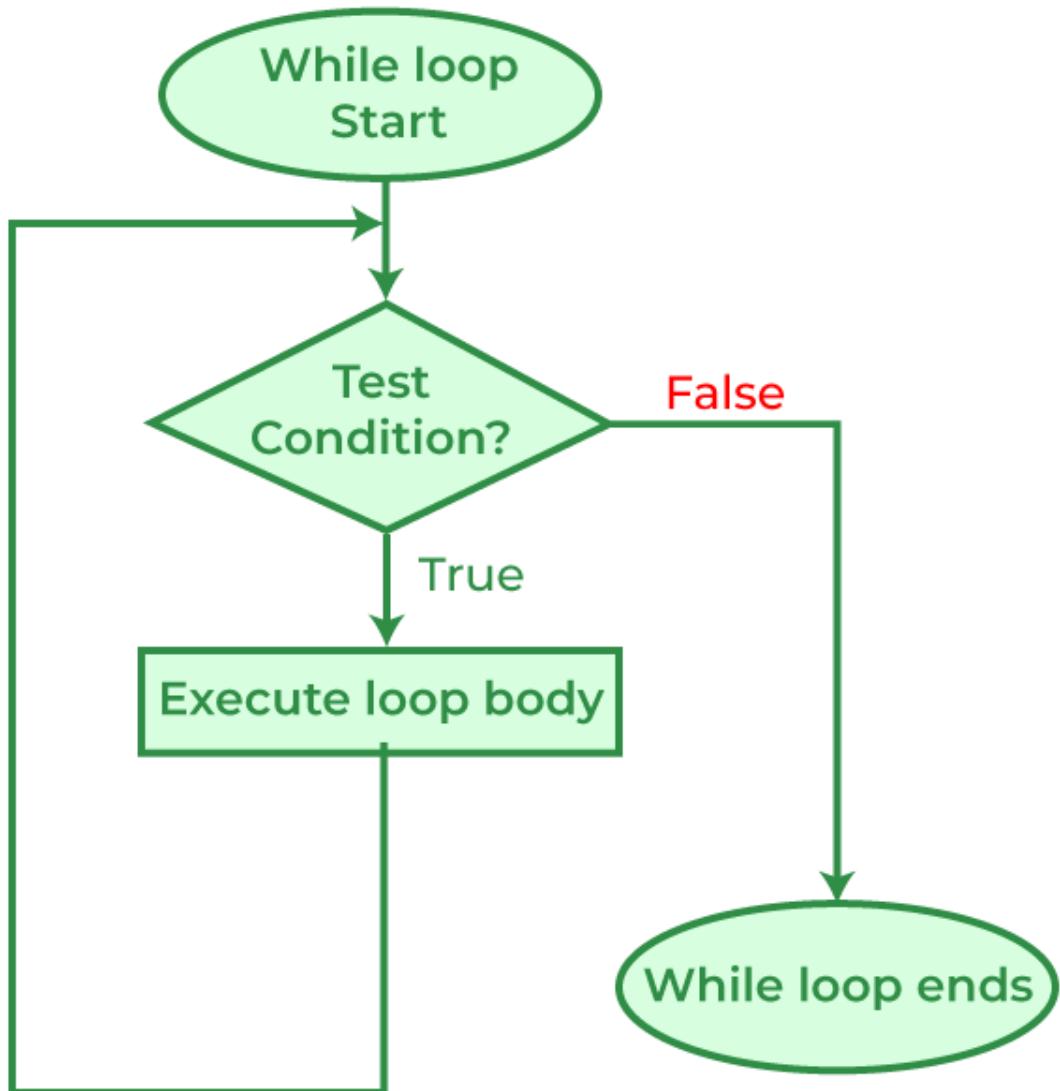
We have already stated that a loop mainly consists of three statements – initialization expression, test expression, and update expression. The syntax of the three loops – For, while, and do while mainly differs in the placement of these three statements.

**Syntax:**

```
initialization expression;
while (test_expression)
{
    // statements
```

```
update_expression;  
}
```

### Flow Diagram of while loop:



### Example:

#### C++

```
// C++ program to Demonstrate while loop  
#include <iostream>  
using namespace std;  
  
int main()
```

```

{
    // initialization expression
    int i = 1;

    // test expression
    while (i < 6) {
        cout << "Hello World\n";

        // update expression
        i++;
    }

    return 0;
}

```

## Output

```

Hello World
Hello World
Hello World
Hello World
Hello World

```

**Time complexity:** O(1)

**Space complexity:** O(1)

It's explanation is same as that of the *for loop*.

## Do-while loop

In Do-while loops also the loop execution is terminated on the basis of test conditions. The main difference between a do-while loop and the while loop is in the do-while loop the condition is tested at the end of the loop body, i.e do-while loop is exit controlled whereas the other two loops are entry-controlled loops.

**Note:** In a do-while loop, the loop body will **execute at least once** irrespective of the test condition.

### Syntax:

```

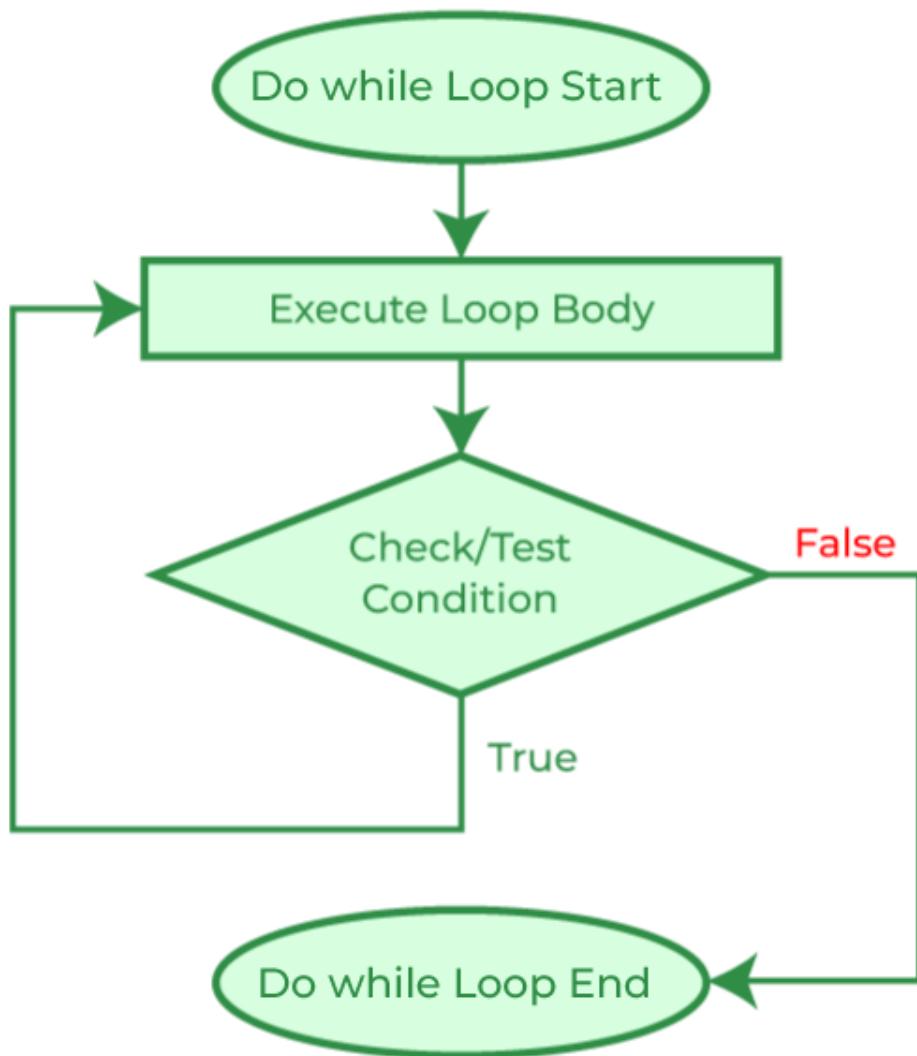
initialization_expression;
do
{
    // statements

    update_expression;
} while (test_expression);

```

**Note:** Notice the semi - colon (";") in the end of loop.

### Flow Diagram of the do-while loop:



### Example:

#### C++

```
// C++ program to Demonstrate do-while loop
#include <iostream>
using namespace std;

int main()
```

```

{
    int i = 2; // Initialization expression

    do {
        // loop body
        cout << "Hello World\n";

        // update expression
        i++;
    } while (i < 1); // test expression

    return 0;
}

```

## Output

Hello World

**Time complexity:** O(1)

**Space complexity:** O(1)

In the above program, the test condition ( $i < 1$ ) evaluates to false. But still, as the loop is an exit – controlled the loop body will execute once.

## What about an Infinite Loop?

An infinite loop (sometimes called an endless loop) is a piece of coding that lacks a **functional exit** so that it repeats indefinitely. An infinite loop occurs when a condition is always evaluated to be true. Usually, this is an error.

## Using For loop:

---

### C++

```

// C++ program to demonstrate infinite loops
// using for and while loop

// Uncomment the sections to see the output
#include <iostream>
using namespace std;
int main()
{
    int i;

    // This is an infinite for loop as the condition
    // expression is blank
    for (;;) {
        cout << "This loop will run forever.\n";
    }
}

```

```

}

// This is an infinite for loop as the condition
// given in while loop will keep repeating infinitely
/*
while (i != 0)
{
    i-- ;
    cout << "This loop will run forever.\n";
}
*/

// This is an infinite for loop as the condition
// given in while loop is "true"
/*
while (true)
{
    cout << "This loop will run forever.\n";
}
*/
}

```

## Output

### Output:

This loop will run forever.

This loop will run forever.

.....

**Time complexity:** O(infinity) as the loop will run forever.

**Space complexity:** O(1)

### Using While loop:

## C++

```

#include <iostream>
using namespace std;

int main()
{
    while (1)
        cout << "This loop will run forever.\n";
    return 0;
}

```

## Output

### Output:

```
This loop will run forever.  
This loop will run forever.  
.....
```

**Time complexity:** O(infinity) as the loop will run forever.

**Space complexity:** O(1)

### Using the Do-While loop:

---

## C++

```
#include <iostream>  
using namespace std;  
  
int main()  
{  
  
    do {  
        cout << "This loop will run forever.\n";  
    } while (1);  
  
    return 0;  
}
```

## Output

### Output:

```
This loop will run forever.  
This loop will run forever.  
.....
```

**Time complexity:** O(infinity) as the loop will run forever.

**Space complexity:** O(1)

## Now let us take a look at decrementing loops.

Sometimes we need to decrement a variable with a looping condition.

### Using for loop

---

#### C++

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

int main() {

    for(int i=5;i>=0;i--){
        cout<<i<<" ";
    }
    return 0;
}
```

#### Output

5 4 3 2 1 0

**Time complexity:** O(1)

**Space complexity:** O(1)

### Using while loop.

---

#### C++

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

int main() {
    //first way is to decrement in the condition itself
    int i=5;
    while(i--){
        cout<<i<<" ";
    }
    cout<<endl;
    //second way is to decrement inside the loop till i is 0
    i=5;
    while(i){
        cout<<i<<" ";
        i--;
    }
}
```

```
    return 0;
}
```

## Output

```
4 3 2 1 0
5 4 3 2 1
```

**Time complexity:** O(1)

**Space complexity:** O(1)

## Using do-while loop

---

### C++

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

int main() {
    int i=5;
    do{
        cout<<i<<" ";
    }while(i--);
}
```

## Output

```
5 4 3 2 1 0
```

**Time complexity:** O(1)

**Space complexity:** O(1)

---

### C++

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

int main() {
    // For loop
    for (int i = 1; i <= 5; i++) {
        cout << "For loop: The value of i is: " << i << endl;
    }

    // While loop
    int j = 1;
```

```
while (j <= 5) {
    cout << "While loop: The value of j is: " << j << endl;
    j++;
}

// Do-while loop
int k = 1;
do {
    cout << "Do-while loop: The value of k is: " << k << endl;
    k++;
} while (k <= 5);

// Range-based for loop
vector<int> myVector = {1, 2, 3, 4, 5};
for (int element : myVector) {
    cout << "Range-based for loop: The value of element is: " << element << endl;
}

return 0;
}
```

## Output

```
For loop: The value of i is: 1
For loop: The value of i is: 2
For loop: The value of i is: 3
For loop: The value of i is: 4
For loop: The value of i is: 5
While loop: The value of j is: 1
While loop: The value of j is: 2
While loop: The value of j is: 3
While loop: The value of j is: 4
While loop: The value of j is: 5
Do-while loop: The value of k is: 1
Do-while loop: The value of k is: 2
Do-while loop: The value of k is: 3
Do-while loop: The value of k is: 4
Do-while loop: The value of k is: 5
Range-based for loop: The value of element is: 1
Range-based for loop: The value of element is: 2
Range-based for loop: The value of element is: 3
Range-based for loop: The value of element is: 4
Range-based for loop: The value of element is: 5
```

---

## C++

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

int main() {

    // for loop example
    cout << "For loop:" << endl;
    for(int i = 0; i < 5; i++) {
        cout << i << endl;
    }

    // while loop example
    cout << "While loop:" << endl;
    int j = 0;
    while(j < 5) {
        cout << j << endl;
        j++;
    }

    // do-while loop example
    cout << "Do-while loop:" << endl;
    int k = 0;
    do {
        cout << k << endl;
        k++;
    } while(k < 5);

    return 0;
}
```

## Output

For loop:

```
0
1
2
3
4
```

While loop:

```
0
1
2
3
4
```

Do-while loop:

```
0
1
2
```

3

4

## Advantages :

1. **High performance:** C++ is a compiled language that can produce efficient and high-performance code. It allows low-level memory manipulation and direct access to system resources, making it ideal for applications that require high performance, such as game development, operating systems, and scientific computing.
2. **Object-oriented programming:** C++ supports object-oriented programming, allowing developers to write modular, reusable, and maintainable code. It provides features such as inheritance, polymorphism, encapsulation, and abstraction that make code easier to understand and modify.
3. **Wide range of applications:** C++ is a versatile language that can be used for a wide range of applications, including desktop applications, games, mobile apps, embedded systems, and web development. It is also used extensively in the development of operating systems, system software, and device drivers.
4. **Standardized language:** C++ is a standardized language, with a specification maintained by the ISO (International Organization for Standardization). This ensures that C++ code written on one platform can be easily ported to another platform, making it a popular choice for cross-platform development.
5. **Large community and resources:** C++ has a large and active community of developers and users, with many resources available online, including documentation, tutorials, libraries, and frameworks. This makes it easy to find help and support when needed.
6. **Interoperability with other languages:** C++ can be easily integrated with other programming languages, such as C, Python, and Java, allowing developers to leverage the strengths of different languages in their applications.

Overall, C++ is a powerful and flexible language that offers many advantages for developers who need to create high-performance, reliable, and scalable applications.

## More Advanced Looping Techniques

- [Range-Based for Loop in C++](#)
- [for each Loop in C++](#)

## Important Points

- Use for a loop when a number of iterations are known beforehand, i.e. the number of times the loop body is needed to be executed is known.

- Use while loops, where an exact number of iterations is not known but the loop termination condition, is known.
- Use do while loop if the code needs to be executed at least once like in Menu-driven programs

## Related Articles:

- [What happens if loop till Maximum of Signed and Unsigned in C/C++?](#)
- [Quiz on Loops](#)

This article is contributed by **Harsh Agarwal**. If you like GeeksforGeeks and would like to contribute, you can also write an article using [write.geeksforgeeks.org](#). See your article appearing on the GeeksforGeeks main page and help other Geeks. Please write comments if you find anything incorrect, or if you want to share more information about the topic discussed above.

635

## Related Articles

1. [Print 1 to 100 in C++ Without Loops and Recursion](#)
2. [Understanding for loops in Java](#)
3. [Nested Loops in C++ with Examples](#)
4. [Loops in Java](#)
5. [How to print N times without using loops or recursion ?](#)
6. [Sum of array Elements without using loops and recursion](#)
7. [Loops and Control Statements \(continue, break and pass\) in Python](#)
8. [main Function in C](#)
9. [printf in C](#)
10. [C++ Error - Does not name a type](#)



SALE!

GeeksforGeeks Courses Upto 25% Off Enroll Now!

[Save 25% on Courses](#) DSA Data Structures Algorithms Interview Preparation Data Science T

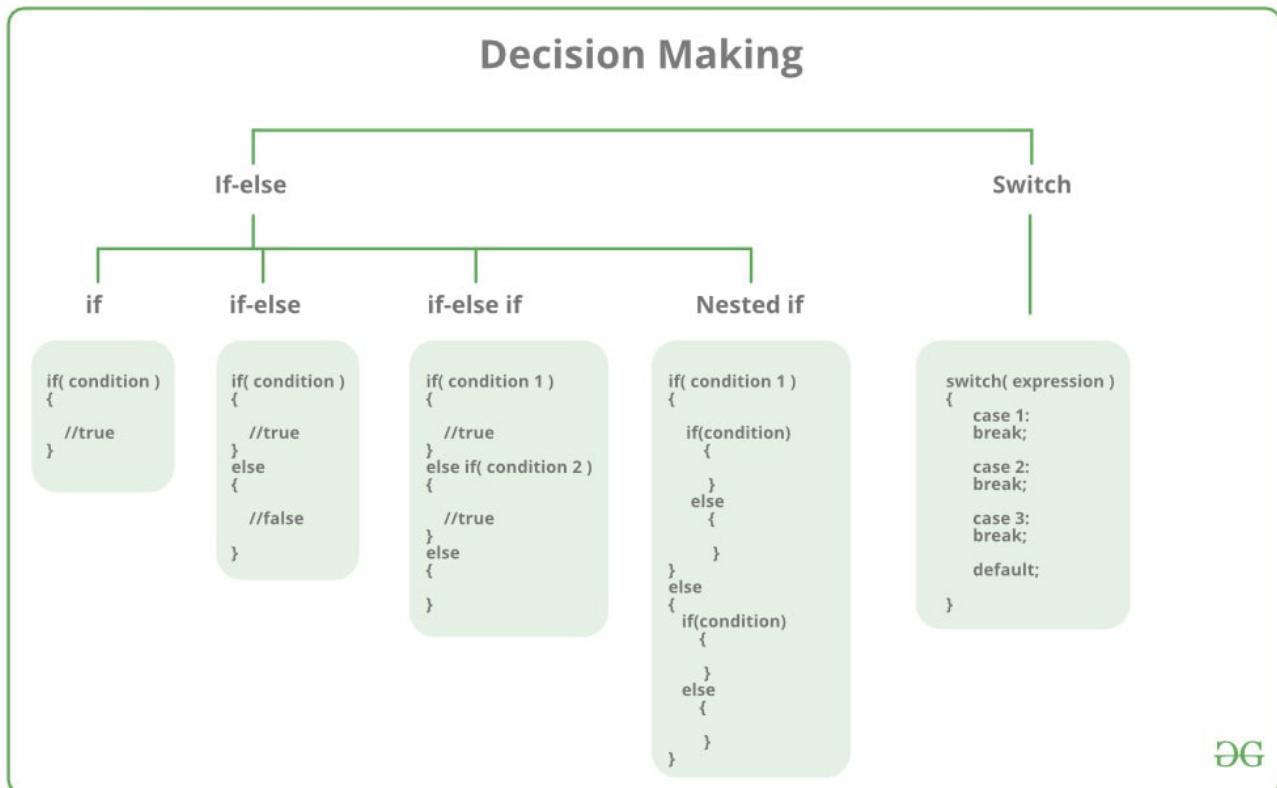
# Decision Making in C / C++ (if , if..else, Nested if, if-else-if )

Difficulty Level : Easy • Last Updated : 16 Jan, 2023

[Read](#)[Discuss](#)[Courses](#)[Practice](#)[Video](#)

There come situations in real life when we need to make some decisions and based on these decisions, we decide what should we do next. Similar situations arise in programming also where we need to make some decisions and based on these decisions we will execute the next block of code.

For example, in C if x occurs then execute y else execute z. There can also be multiple conditions like in C if x occurs then execute p, else if condition y occurs execute q, else execute r. This condition of C else-if is one of the many ways of importing multiple conditions. The Decision Making Statements are used to evaluate the one or more conditions and make the decision whether to execute set of statement or not.



DG

Decision-making statements in programming languages decide the direction of the flow of program execution. Decision-making statements available in C or C++ are:

1. if statement
2. if-else statements
3. nested if statements
4. if-else-if ladder
5. switch statements
6. Jump Statements:
  - break
  - continue
  - goto
  - return

## 1. if statement in C/C++

if statement is the most simple decision-making statement. It is used to decide whether a certain statement or block of statements will be executed or not i.e if a certain condition is true then a block of statement is executed otherwise not.

### Syntax:

AD

```
if(condition)
{
    // Statements to execute if
    // condition is true
}
```

Here, the **condition** after evaluation will be either true or false. C if statement accepts boolean values – if the value is true then it will execute the block of statements below it otherwise not. If we do not provide the curly braces '{' and '}' after if(condition) then by default if statement will consider the first immediately below statement to be inside its block.

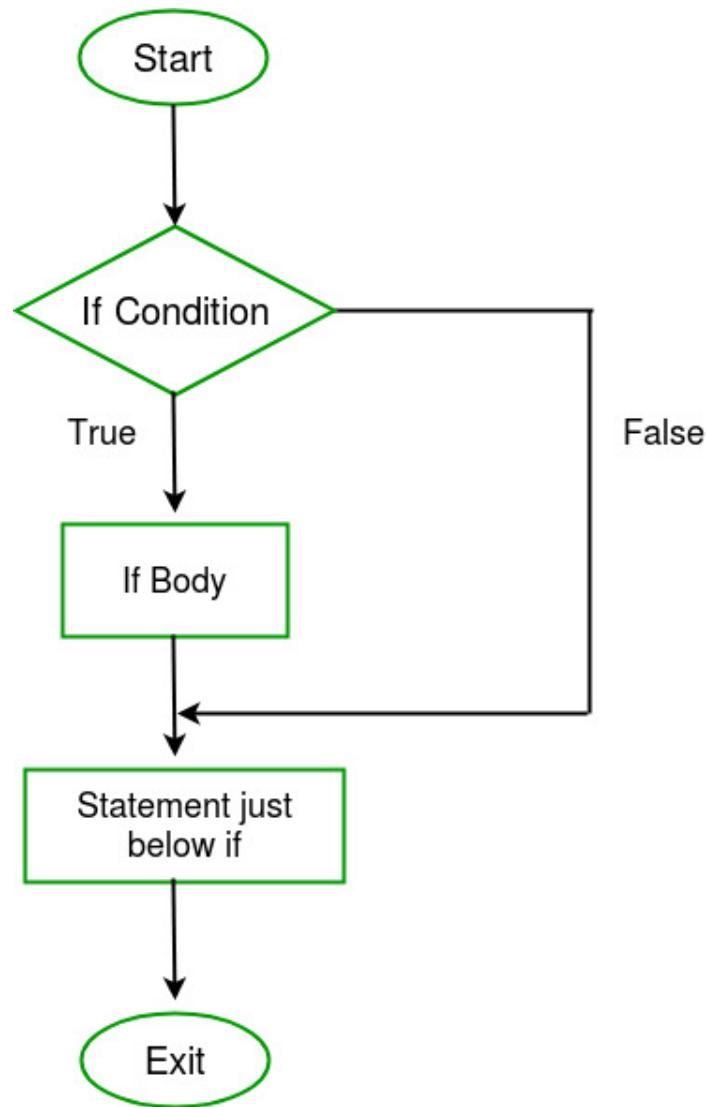
### Example:

If Ram can having 100 GeekBits then he can redeem these GeekBits and get the GFG T-shirt

```
if(condition)
    statement1;
    statement2;

// Here if the condition is true, if block
// will consider only statement1 to be inside
// its block.
```

### Flowchart



## C

```
// C program to illustrate If statement
#include <stdio.h>

int main()
{
    int i = 10;

    if (i > 15) {
        printf("10 is greater than 15");
    }

    printf("I am Not in if");
}
```

## C++

```
// C++ program to illustrate If statement
```

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

int main()
{
    int i = 10;

    if (i > 15) {
        cout << "10 is greater than 15";
    }

    cout << "I am Not in if";
}
```

**Output:**

I am Not in if

As the condition present in the if statement is false. So, the block below the if statement is not executed.

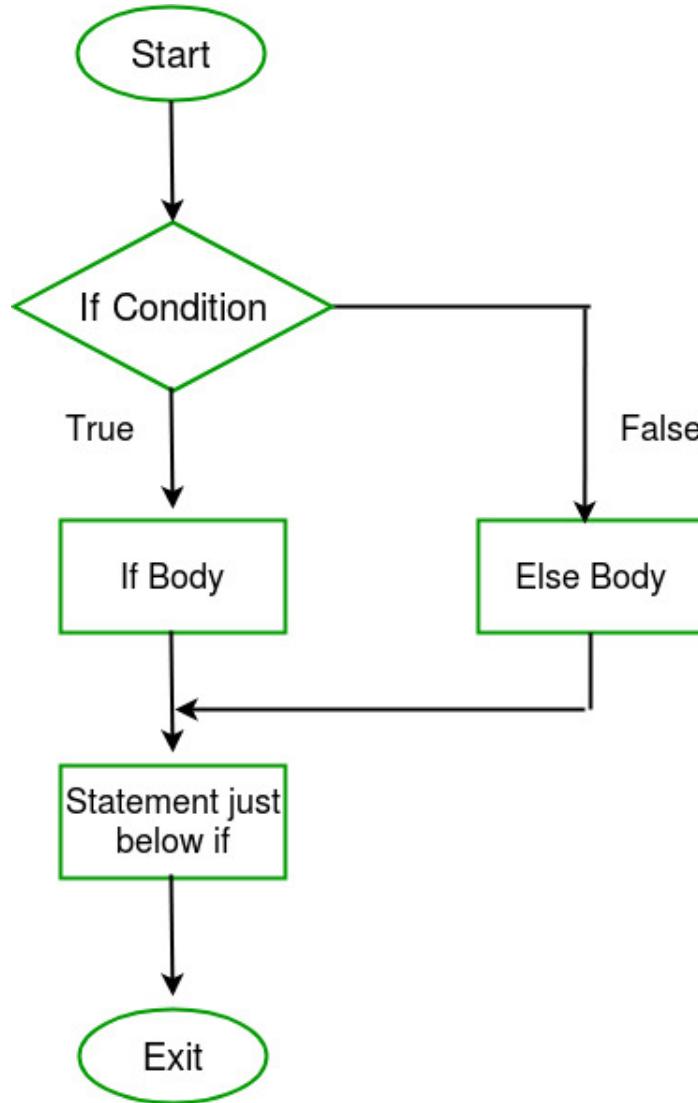
**2. if-else in C/C++**

The *if* statement alone tells us that if a condition is true it will execute a block of statements and if the condition is false it won't. But what if we want to do something else if the condition is false. Here comes the C *else* statement. We can use the *else* statement with the *if* statement to execute a block of code when the condition is false.

**Syntax:**

```
if (condition)
{
    // Executes this block if
    // condition is true
}
else
{
    // Executes this block if
    // condition is false
}
```

**Flowchart:**



### Example:

The person who having correct 50 Geek Bits is redeem the gifts otherwise they can't redeem.

### C

```

// C program to illustrate If statement
#include <stdio.h>

int main()
{
    int i = 20;

    if (i < 15) {

        printf("i is smaller than 15");
    }
    else {

        printf("i is greater than 15");
    }
}

```

```
    return 0;
}
```

## C++

```
// C++ program to illustrate if-else statement
#include <iostream>
using namespace std;

int main()
{
    int i = 20;

    if (i < 15)
        cout << "i is smaller than 15";
    else
        cout << "i is greater than 15";

    return 0;
}
```

### Output:

i is greater than 15

The block of code following the `else` statement is executed as the condition present in the `if` statement is false.

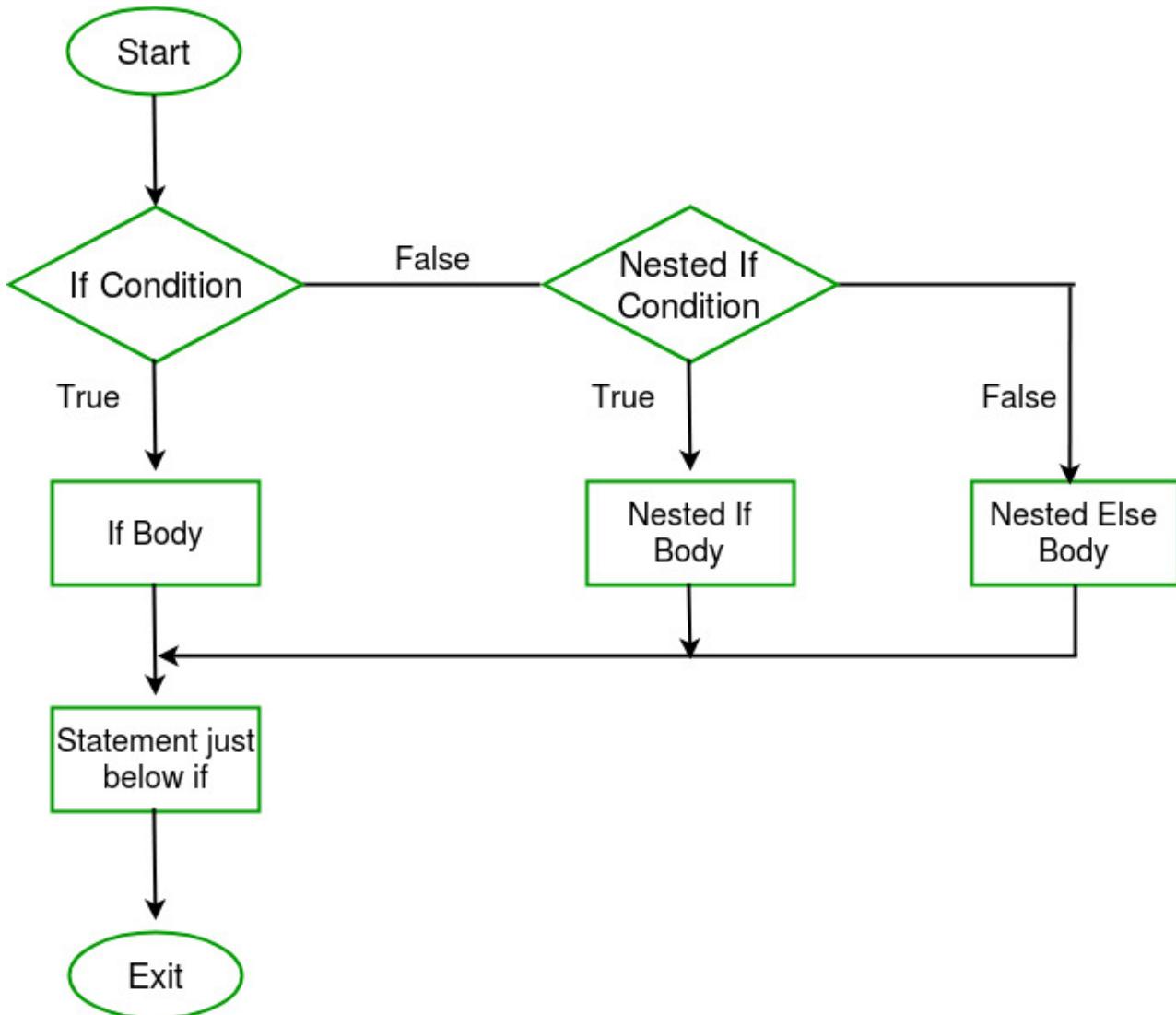
## 3. nested-if in C/C++

A nested if in C is an if statement that is the target of another if statement. Nested if statements mean an if statement inside another if statement. Yes, both C and C++ allow us to nested if statements within if statements, i.e, we can place an if statement inside another if statement.

### Syntax:

```
if (condition1)
{
    // Executes when condition1 is true
    if (condition2)
    {
        // Executes when condition2 is true
    }
}
```

## Flowchart



### Example:

If the person having more than 50 Geek Bits and less than 150 Geek Bits he won the GFG T-shirt.

## C

```

// C program to illustrate nested-if statement
#include <stdio.h>

int main()
{
    int i = 10;

    if (i == 10) {
        // First if statement
        if (i < 15)
            printf("i is smaller than 15\n");
    }
}
  
```

```

// Nested - if statement
// Will only be executed if statement above
// is true
if (i < 12)
    printf("i is smaller than 12 too\n");
else
    printf("i is greater than 15");
}

return 0;
}

```

**C++**

```

// C++ program to illustrate nested-if statement
#include <iostream>
using namespace std;

int main()
{
    int i = 10;

    if (i == 10) {
        // First if statement
        if (i < 15)
            cout << "i is smaller than 15\n";

        // Nested - if statement
        // Will only be executed if
        // statement above is true
        if (i < 12)
            cout << "i is smaller than 12 too\n";
        else
            cout << "i is greater than 15";
    }

    return 0;
}

```

**Output:**

```

i is smaller than 15
i is smaller than 12 too

```

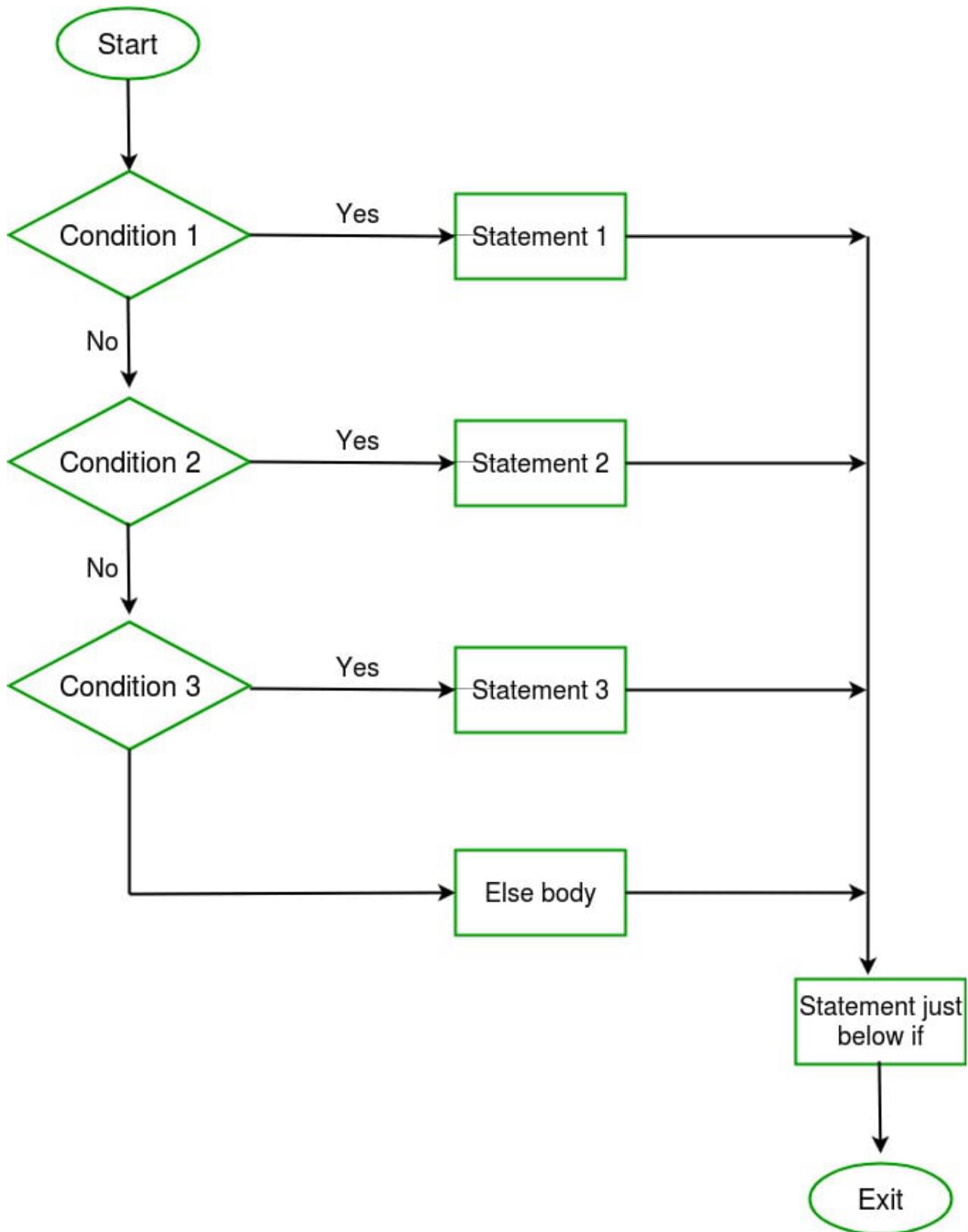
**4. if-else-if ladder in C/C++**

Here, a user can decide among multiple options. The C if statements are executed from the top down. As soon as one of the conditions controlling the if is true, the statement associated with that if is executed, and the rest of the C else-if ladder is bypassed. If none

of the conditions is true, then the final else statement will be executed. if-else-if ladder is similar to switch statement.

### Syntax:

```
if (condition)
    statement;
else if (condition)
    statement;
.
.
else
    statement;
```

**Example:**

If the person having the 50 Geek Bits then he get the GfG Course Coupon otherwise he will having 100 Geek Bits then he get the GfG T-shirt otherwise he will having the 200 Geek Bits then he get the GfG Bag otherwise he will having less than 50 he cant get anything.

**C**

```
// C program to illustrate nested-if statement
#include <stdio.h>

int main()
{
    int i = 20;

    if (i == 10)
        printf("i is 10");
    else if (i == 15)
        printf("i is 15");
    else if (i == 20)
        printf("i is 20");
    else
        printf("i is not present");
}
```

## C++

```
// C++ program to illustrate if-else-if ladder
#include <iostream>
using namespace std;

int main()
{
    int i = 20;

    if (i == 10)
        cout << "i is 10";
    else if (i == 15)
        cout << "i is 15";
    else if (i == 20)
        cout << "i is 20";
    else
        cout << "i is not present";
}
```

## Output:

i is 20

## 5. Jump Statements in C/C++

These statements are used in C or C++ for the unconditional flow of control throughout the functions in a program. They support four types of jump statements:

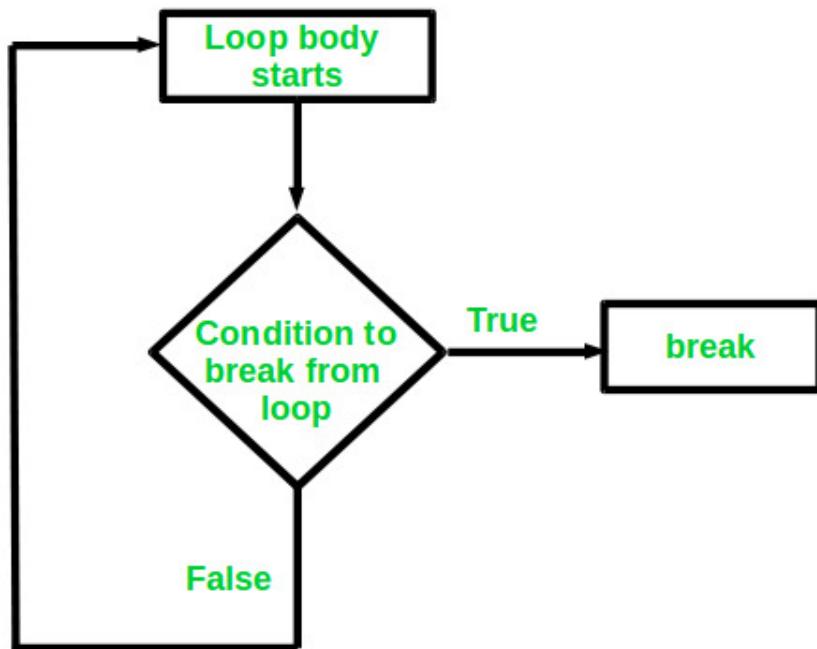
### A) break

This loop control statement is used to terminate the loop. As soon as the [break](#) statement is encountered from within a loop, the loop iterations stop there, and control returns from the loop immediately to the first statement after the loop.

### Syntax:

```
break;
```

Basically, break statements are used in situations when we are not sure about the actual number of iterations for the loop or we want to terminate the loop based on some condition.



### Example:

#### C

```
// C program to illustrate
// to show usage of break
// statement
#include <stdio.h>

void findElement(int arr[], int size, int key)
{
    // loop to traverse array and search for key
    for (int i = 0; i < size; i++) {
        if (arr[i] == key) {
            printf("Element found at position: %d",
                   (i + 1));
            break;
        }
    }
}
```

```

}

int main()
{
    int arr[] = { 1, 2, 3, 4, 5, 6 };

    // no of elements
    int n = 6;

    // key to be searched
    int key = 3;

    // Calling function to find the key
    findElement(arr, n, key);

    return 0;
}

```

## C++

```

// C++ program to illustrate
// to show usage of break
// statement
#include <iostream>
using namespace std;

void findElement(int arr[], int size, int key)
{
    // loop to traverse array and search for key
    for (int i = 0; i < size; i++) {
        if (arr[i] == key) {
            cout << "Element found at position: "
                << (i + 1);
            break;
        }
    }
}

// Driver program to test above function
int main()
{
    int arr[] = { 1, 2, 3, 4, 5, 6 };
    int n = 6; // no of elements
    int key = 3; // key to be searched

    // Calling function to find the key
    findElement(arr, n, key);

    return 0;
}

```

## Output:

Element found at position: 3

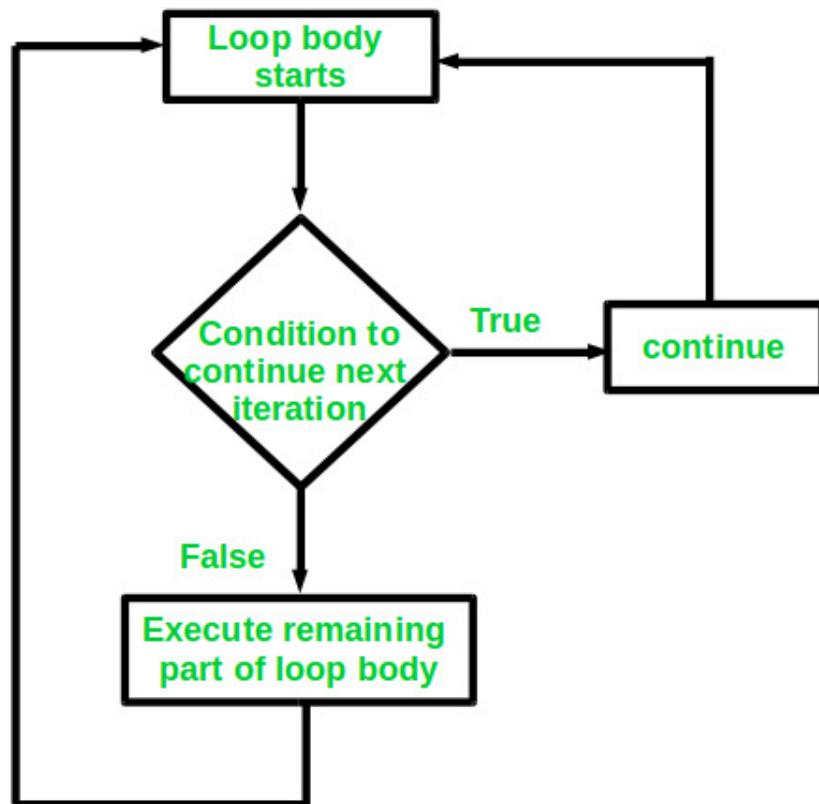
## B) continue

This loop control statement is just like the [break statement](#). The [continue statement](#) is opposite to that of the break *statement*, instead of terminating the loop, it forces to execute the next iteration of the loop.

As the name suggests the continue statement forces the loop to continue or execute the next iteration. When the continue statement is executed in the loop, the code inside the loop following the continue statement will be skipped and the next iteration of the loop will begin.

### Syntax:

```
continue;
```



### Example:

## C

```
// C program to explain the use
// of continue statement
#include <stdio.h>
```

```

int main()
{
    // loop from 1 to 10
    for (int i = 1; i <= 10; i++) {

        // If i is equals to 6,
        // continue to next iteration
        // without printing
        if (i == 6)
            continue;

        else
            // otherwise print the value of i
            printf("%d ", i);
    }

    return 0;
}

```

## C++

```

// C++ program to explain the use
// of continue statement

#include <iostream>
using namespace std;

int main()
{
    // loop from 1 to 10
    for (int i = 1; i <= 10; i++) {

        // If i is equals to 6,
        // continue to next iteration
        // without printing
        if (i == 6)
            continue;

        else
            // otherwise print the value of i
            cout << i << " ";
    }

    return 0;
}

```

## Output:

1 2 3 4 5 7 8 9 10

If you create a variable in if-else in C/C++, it will be local to that if/else block only. You can use global variables inside the if/else block. If the name of the variable you created in

if/else is as same as any global variable then priority will be given to `local variable`.

## C

```
#include <stdio.h>

int main()
{
    int gfg = 0; // local variable for main
    printf("Before if-else block %d\n", gfg);
    if (1) {
        int gfg = 100; // new local variable of if block
        printf("if block %d\n", gfg);
    }
    printf("After if block %d", gfg);
    return 0;
}
```

## C++

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

int main()
{
    int gfg = 0; // local variable for main
    cout << "Before if-else block " << gfg << endl;
    if (1) {
        int gfg = 100; // new local variable of if block
        cout << "if block " << gfg << endl;
    }
    cout << "After if block " << gfg << endl;
    return 0;
}
/*
 Before if-else block 0
 if block 100
 After if block 0
*/
```

## Output:

```
Before if-else block 0
if block 100
After if block 0
```

## C) goto

The [goto statement](#) in C/C++ also referred to as the unconditional jump statement can be used to jump from one point to another within a function.

### Syntax:

Syntax1		Syntax2
---------	--	---------

---

goto label;		label:
-------------	--	--------

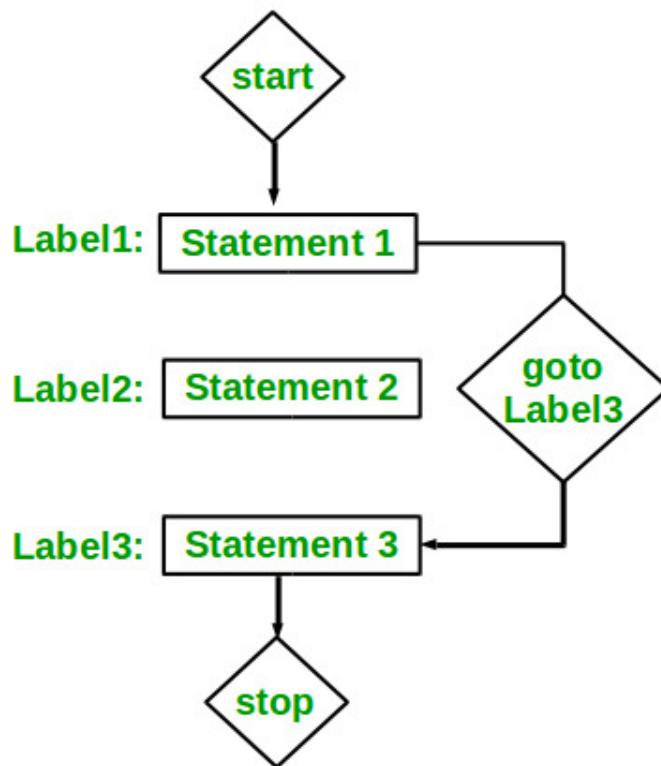
.		.
---	--	---

.		.
---	--	---

.		.
---	--	---

label:		goto label;
--------	--	-------------

In the above syntax, the first line tells the compiler to go to or jump to the statement marked as a label. Here, a label is a user-defined identifier that indicates the target statement. The statement immediately followed after 'label:' is the destination statement. The 'label:' can also appear before the 'goto label;' statement in the above syntax.



### Examples:

#### C

```
// C program to print numbers
// from 1 to 10 using goto
// statement
#include <stdio.h>
```

```
// function to print numbers from 1 to 10
void printNumbers()
{
    int n = 1;
label:
    printf("%d ", n);
    n++;
    if (n <= 10)
        goto label;
}

// Driver program to test above function
int main()
{
    printNumbers();
    return 0;
}
```

## C++

```
// C++ program to print numbers
// from 1 to 10 using goto
// statement
#include <iostream>
using namespace std;

// function to print numbers from 1 to 10
void printNumbers()
{
    int n = 1;
label:
    cout << n << " ";
    n++;
    if (n <= 10)
        goto label;
}

// Driver program to test above function
int main()
{
    printNumbers();
    return 0;
}
```

## Output:

1 2 3 4 5 6 7 8 9 10

## D) return

The [return](#) in C or C++ returns the flow of the execution to the function from where it is called. This statement does not mandatorily need any conditional statements. As soon as the statement is executed, the flow of the program stops immediately and returns the control from where it was called. The return statement may or may not return anything for a void function, but for a non-void function, a return value must be returned.

### Syntax:

```
return[expression];
```

### Example:

---

## C

```
// C code to illustrate return
// statement
#include <stdio.h>

// non-void return type
// function to calculate sum
int SUM(int a, int b)
{
    int s1 = a + b;
    return s1;
}

// returns void
// function to print
void Print(int s2)
{
    printf("The sum is %d", s2);
    return;
}

int main()
{
    int num1 = 10;
    int num2 = 10;
    int sum_of = SUM(num1, num2);
    Print(sum_of);
    return 0;
}
```

## C++

```
// C++ code to illustrate return
// statement
#include <iostream>
using namespace std;
```

```
// non-void return type
// function to calculate sum
int SUM(int a, int b)
{
    int s1 = a + b;
    return s1;
}

// returns void
// function to print
void Print(int s2)
{
    cout << "The sum is " << s2;
    return;
}

int main()
{
    int num1 = 10;
    int num2 = 10;
    int sum_of = SUM(num1, num2);
    Print(sum_of);
    return 0;
}
```

## Output:

The sum is 20

This article is contributed by **Harsh Agarwal**. If you like GeeksforGeeks and would like to contribute, you can also write an article using [write.geeksforgeeks.org](https://www.geeksforgeeks.org/write-for-us/) or mail your article to review-team@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

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

363

## Related Articles

1. [C program to invert \(making negative\) an image content in PGM format](#)

---

2. [Making your own Linux Shell in C](#)

---

3. [Publicly inherit a base class but making some of public method as private](#)

---

4. [Nested Classes in C++](#)

**SALE!**  
**GeeksforGeeks Courses Upto 25% Off Enroll Now!**



Save 25% on Courses   DSA   Data Structures   Algorithms   Interview Preparation   Data Science   T

# I/O Redirection in C++

Difficulty Level : Medium • Last Updated : 22 Jun, 2022

[Read](#)   [Discuss](#)   [Courses](#)   [Practice](#)   [Video](#)

In C, we could use the function [freopen\(\)](#) to redirect an existing FILE pointer to another stream. The prototype for freopen() is given as

```
FILE * freopen ( const char * filename, const char * mode, FILE * stream );
```

For Example, to redirect the stdout to say a textfile, we could write :

```
freopen ("text_file.txt", "w", stdout);
```

While this method is still supported in C++, this article discusses another way to redirect I/O streams.

C++ being an object-oriented programming language, gives us the ability to not only define our own streams but also redirect standard streams. Thus, in C++, a stream is an object whose behavior is defined by a class. Thus, anything that behaves like a stream is also a stream.

## Streams Objects in C++ are mainly of three types :

- **istream** : Stream object of this type can only perform input operations from the stream
- **ostream** : These objects can only be used for output operations.
- **iostream** : Can be used for both input and output operations

All these classes, as well as file stream classes, derived from the classes: ios and streambuf. Thus, ifstream and IO stream objects behave similarly.

All stream objects also have an associated data member of class streambuf. Simply put, streambuf object is the buffer for the stream. When we read data from a stream, we don't read it directly from the source, but instead, we read it from the buffer which is linked to the

source. Similarly, output operations are first performed on the buffer, and then the buffer is flushed (written to the physical device) when needed.

C++ allows us to set the stream buffer for any stream, So the task of redirecting the stream simply reduces to changing the stream buffer associated with the stream. Thus, to redirect a Stream A to Stream B we need to do:-

AD

1. Get the stream buffer of A and store it somewhere
2. Set the stream buffer of A to the stream buffer of B
3. If needed to reset the stream buffer of A to its previous stream buffer

We can use the function [`ios::rdbuf\(\)`](#) to perform below two operations.

- 1) `stream_object.rdbuf()`: Returns pointer to the stream buffer of `stream_object`
- 2) `stream_object.rdbuf(streambuf * p)`: Sets the stream buffer to the object pointed by `p`

Here is an example program below to show the steps

## CPP

```
// Cpp program to redirect cout to a file
#include <fstream>
#include <iostream>
#include <string>

using namespace std;

int main()
{
    fstream file;
    file.open("cout.txt", ios::out);
    string line;

    // Backup streambuffers of cout
    streambuf* stream_buffer_cout = cout.rdbuf();
    streambuf* stream_buffer_in = cin.rdbuf();
```

```

// Get the streambuffer of the file
streambuf* stream_buffer_file = file.rdbuf();

// Redirect cout to file
cout.rdbuf(stream_buffer_file);

cout << "This line written to file" << endl;

// Redirect cout back to screen
cout.rdbuf(stream_buffer_cout);
cout << "This line is written to screen" << endl;

file.close();
return 0;
}

```

Output:

```

This line is written to screen
Contents of file cout.txt:
This line written to file

```

**Time Complexity: O(1)**

**Space Complexity: O(1)**

**Note:**

The above steps can be condensed into a single step

```

auto cout_buf = cout.rdbuf(file.rdbuf())

// sets cout's streambuffer and returns the old
// streambuffer back to cout_buf

```

References:

[CPP IOS](#)

138

## Related Articles

1. [C++ Error - Does not name a type](#)

---

2. [Execution Policy of STL Algorithms in Modern C++](#)

**SALE!**  
**GeeksforGeeks Courses Upto 25% Off Enroll Now!**



Save 25% on Courses   DSA   Data Structures   Algorithms   Interview Preparation   Data Science   T

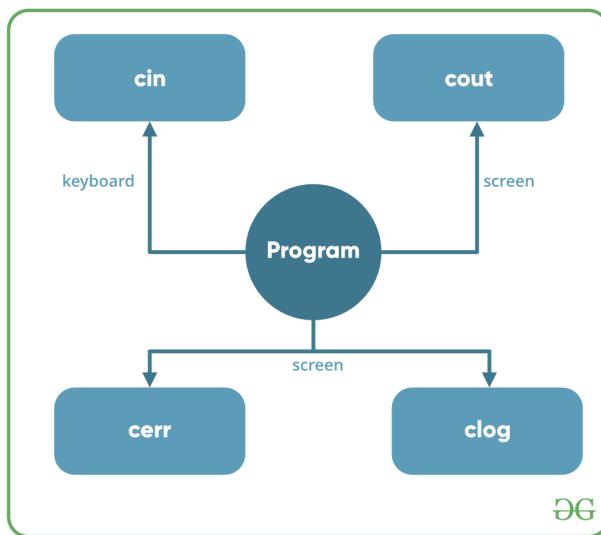
# Basic Input / Output in C++

Difficulty Level : Easy • Last Updated : 25 Jan, 2023

[Read](#)   [Discuss](#)   [Courses](#)   [Practice](#)   [Video](#)

C++ comes with libraries that provide us with many ways for performing input and output. In C++ input and output are performed in the form of a sequence of bytes or more commonly known as **streams**.

- **Input Stream:** If the direction of flow of bytes is from the device (for example, Keyboard) to the main memory then this process is called input.
- **Output Stream:** If the direction of flow of bytes is opposite, i.e. from main memory to device ( display screen ) then this process is called output.



**Header files available in C++ for Input/Output operations are:**

1. **iostream:** iostream stands for standard input-output stream. This header file contains definitions of objects like `cin`, `cout`, `cerr`, etc.
2. **iomanip:** iomanip stands for input-output manipulators. The methods declared in these files are used for manipulating streams. This file contains definitions of `setw`,

setprecision, etc.

3. **fstream**: This header file mainly describes the file stream. This header file is used to handle the data being read from a file as input or data being written into the file as output.
4. **bits/stdc++**: This header file includes every standard library. In programming contests, using this file is a good idea, when you want to reduce the time wasted in doing chores; especially when your rank is time sensitive. To know more about this header file refer [this article](#).

In C++ after the header files, we often use '*using namespace std*'. The reason behind it is that all of the standard library definitions are inside the namespace std. As the library functions are not defined at global scope, so in order to use them we use *namespace std*. So, that we don't need to write STD:: at every line (eg. STD::cout etc.). To know more refer [this article](#).

AD

The two instances **cout in C++** and **cin in C++** of iostream class are used very often for printing outputs and taking inputs respectively. These two are the most basic methods of taking input and printing output in C++. To use cin and cout in C++ one must include the header file *iostream* in the program.

This article mainly discusses the objects defined in the header file *iostream* like the cin and cout.

- **Standard output stream (cout)**: Usually the standard output device is the display screen. The C++ **cout** statement is the instance of the ostream class. It is used to produce output on the standard output device which is usually the display screen. The data needed to be displayed on the screen is inserted in the standard output stream (cout) using the insertion operator(**<<**).

## C++

```
#include <iostream>

using namespace std;
```

```

int main()
{
    char sample[] = "GeeksforGeeks";

    cout << sample << " - A computer science portal for geeks";

    return 0;
}

```

**Output:**

GeeksforGeeks - A computer science portal for geeks

**Time Complexity:** O(1)

**Auxiliary Space:** O(1)

In the above program, the insertion operator(`<<`) inserts the value of the string variable **sample** followed by the string "A computer science portal for geeks" in the standard output stream **cout** which is then displayed on the screen.

- **standard input stream (cin):** Usually the input device in a computer is the keyboard. C++ **cin** statement is the instance of the class **istream** and is used to read input from the standard input device which is usually a keyboard.

The extraction operator(`>>`) is used along with the object **cin** for reading inputs. The extraction operator extracts the data from the object **cin** which is entered using the keyboard.

---

**C++**

```

#include <iostream>
using namespace std;

int main()
{
    int age;

    cout << "Enter your age:";
    cin >> age;
    cout << "\nYour age is: " << age;

    return 0;
}

```

**Input :**

**Output:**

```
Enter your age:
```

```
Your age is: 18
```

**Time Complexity:** O(1)

**Auxiliary Space:** O(1)

The above program asks the user to input the age. The object `cin` is connected to the input device. The age entered by the user is extracted from `cin` using the extraction operator (`>>`) and the extracted data is then stored in the variable `age` present on the right side of the extraction operator.

- **Un-buffered standard error stream (cerr):** The C++ `cerr` is the standard error stream that is used to output the errors. This is also an instance of the `iostream` class. As `cerr` in C++ is un-buffered so it is used when one needs to display the error message immediately. It does not have any buffer to store the error message and display it later.
- The main difference between `cerr` and `cout` comes when you would like to redirect output using "cout" that gets redirected to file if you use "cerr" the error doesn't get stored in file.(This is what un-buffered means ..It cant store the message)

**C++**

```
#include <iostream>

using namespace std;

int main()
{
    cerr << "An error occurred";
    return 0;
}
```

**Output:**

```
An error occurred
```

**Time Complexity:** O(1)

**Auxiliary Space:** O(1)

- **buffered standard error stream (clog):** This is also an instance of `ostream` class and used to display errors but unlike `cerr` the error is first inserted into a buffer and is stored in the buffer until it is not fully filled. or the buffer is not explicitly flushed (using `flush()`). The error message will be displayed on the screen too.

## C++

```
#include <iostream>

using namespace std;

int main()
{
    clog << "An error occurred";

    return 0;
}
```

### Output:

An error occurred

**Time Complexity:** O(1)

**Auxiliary Space:** O(1)

C++ Programming Language Tutorial | Basic Input / Output in C++ | Ge...



### Related Articles:

- [cout << endl vs cout << "\n" in C++](#)
- [Problem with scanf\(\) when there is fgets\(\)/gets\(\)/scanf\(\) after it](#)
- [How to use getline\(\) in C++ when there are blank lines in input?](#)
- [Cin-Cout vs Scanf-Printf](#)

This article is contributed by [Harsh Agarwal](#). If you like GeeksforGeeks and would like to contribute, you can also write an article using [write.geeksforgeeks.org](#) or mail your article to review-team@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

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

SALE!  
GeeksforGeeks Courses Upto 25% Off Enroll Now!

[Save 25% on Courses](#)[DSA](#)[Data Structures](#)[Algorithms](#)[Interview Preparation](#)[Data Science](#)[T](#)

# Clearing The Input Buffer In C/C++

Difficulty Level : Medium • Last Updated : 30 Oct, 2022

[Read](#)[Discuss\(20+\)](#)[Courses](#)[Practice](#)[Video](#)

## What is a buffer?

A temporary storage area is called a buffer. All standard input and output devices contain an input and output buffer. In standard C/C++, streams are buffered. For example, in the case of standard input, when we press the key on the keyboard, it isn't sent to your program, instead of that, it is sent to the buffer by the operating system, till the time is allotted to that program.

## How does it affect Programming?

On various occasions, you may need to clear the unwanted buffer so as to get the next input in the desired container and not in the buffer of the previous variable. For example, in the case of C after encountering "scanf()", if we need to input a character array or character, and in the case of C++, after encountering the "cin" statement, we require to input a character array or a string, we require to clear the input buffer or else the desired input is occupied by a buffer of the previous variable, not by the desired container. On pressing "Enter" (carriage return) on the output screen after the first input, as the buffer of the previous variable was the space for a new container(as we didn't clear it), the program skips the following input of the container.

AD

## In the case of C Programming.

---

### C

```
// C Code to explain why not
// clearing the input buffer
// causes undesired outputs
#include <stdio.h>
int main()
{
    char str[80], ch;

    // Scan input from user -
    // GeeksforGeeks for example
    scanf("%s", str);

    // Scan character from user-
    // 'a' for example
    ch = getchar();

    // Printing character array,
    // prints "GeeksforGeeks")
    printf("%s\n", str);

    // This does not print
    // character 'a'
    printf("%c", ch);

    return 0;
}
```

Input:

```
GeeksforGeeks
a
```

Output:

```
GeeksforGeeks
```

**Time Complexity: O(1)**

## In the case of C++

---

### C++

```
// C++ Code to explain why
// not clearing the input
// buffer causes undesired
```

```
// outputs
#include<iostream>
#include<vector>
using namespace std;

int main()
{
    int a;
    char ch[80];

    // Enter input from user
    // - 4 for example
    cin >> a;

    // Get input from user -
    // "GeeksforGeeks" for example
    cin.getline(ch,80);

    // Prints 4
    cout << a << endl;

    // Printing string : This does
    // not print string
    cout << ch << endl;

    return 0;
}
```

Input:

```
4
GeeksforGeeks
```

Output:

```
4
```

## Time Complexity: O(1)

In both of the above codes, the output is not printed as desired. The reason for this is an occupied Buffer. The "\n" character goes remains there in the buffer and is read as the next input.

### How can it be resolved?

#### In the case of C:

**1. Using “while ((getchar()) != '\n');”:** Typing “while ((getchar()) != '\n');” reads the buffer characters till the end and discards them(including newline) and using it after the “scanf()” statement clears the input buffer and allows the input in the desired container.

## C

```
// C Code to explain why adding
// "while ( (getchar()) != '\n');"
// after "scanf()" statement
// flushes the input buffer
#include<stdio.h>

int main()
{
    char str[80], ch;

    // scan input from user -
    // GeeksforGeeks for example
    scanf("%s", str);

    // flushes the standard input
    // (clears the input buffer)
    while ((getchar()) != '\n');

    // scan character from user -
    // 'a' for example
    ch = getchar();

    // Printing character array,
    // prints "GeeksforGeeks")
    printf("%s\n", str);

    // Printing character a: It
    // will print 'a' this time
    printf("%c", ch);

    return 0;
}
```

Input:

GeeksforGeeks  
a

Output:

GeeksforGeeks  
a

**Time Complexity: O(n)**, where n is the size of the string.

**2. Using "fflush(stdin)":** Typing "fflush(stdin)" after "scanf()" statement, also clears the input buffer but generally it's use is avoided and is termed to be "undefined" for input

stream as per the C++11 standards.

### In the case of C++:

#### 1. Using " cin.ignore(numeric\_limits::max(),'\\n'); " :- Typing

"cin.ignore(numeric\_limits::max(),'\\n');" after the "cin" statement discards everything in the input stream including the newline.

---

## C++

```
// C++ Code to explain how
// "cin.ignore(numeric_limits
// <streamsize>::max(),'\\n');"
// discards the input buffer
#include <iostream>

// for <streamsize>
#include <ios>

// for numeric_limits
#include <limits>
using namespace std;

int main()
{
    int a;
    char str[80];

    // Enter input from user
    // - 4 for example
    cin >> a;

    // discards the input buffer
    cin.ignore(numeric_limits<streamsize>::max(), '\\n');

    // Get input from user -
    // GeeksforGeeks for example
    cin.getline(str, 80);

    // Prints 4
    cout << a << endl;

    // Printing string : This
    // will print string now
    cout << str << endl;

    return 0;
}
```

Input:

4

GeeksforGeeks

Output:

4

GeeksforGeeks

### Time Complexity: O(1)

**2. Using " cin.sync() ":** Typing "cin.sync()" after the "cin" statement discards all that is left in the buffer. Though "cin.sync()" **does not work** in all implementations (According to C++11 and above standards).

---

## C++

```
// C++ Code to explain how " cin.sync(); "
// discards the input buffer
#include <iostream>
#include <limits>

using namespace std;

int main()
{
    int a;
    char str[80];

    // Enter input from user
    // - 4 for example
    cin >> a;

    // Discards the input buffer
    cin.sync();

    // Get input from user -
    // GeeksforGeeks for example
    cin.getline(str, 80);

    // Prints 4
    cout << a << endl;

    // Printing string - this
    // will print string now
    cout << str << endl;

    return 0;
}
```

Input:

```
4
GeeksforGeeks
```

Output:

```
4
```

### Time Complexity: O(1)

**3. Using " cin >> ws ":** Typing "cin>>ws" after "cin" statement tells the compiler to ignore buffer and also to discard all the whitespaces before the actual content of string or character array.

---

## C++

```
// C++ Code to explain how "cin >> ws"
// discards the input buffer along with
// initial white spaces of string

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

int main()
{
    int a;
    string s;

    // Enter input from user -
    // 4 for example
    cin >> a;

    // Discards the input buffer and
    // initial white spaces of string
    cin >> ws;

    // Get input from user -
    // GeeksforGeeks for example
    getline(cin, s);

    // Prints 4 and GeeksforGeeks :
    // will execute print a and s
    cout << a << endl;
    cout << s << endl;

    return 0;
}
```

Input:

```
4
GeeksforGeeks
```

Output:

```
4
GeeksforGeeks
```

### Time Complexity: O(1)

**4. Using "fflush(stdin)":** Typing "fflush(stdin)" after taking the input stream by "cin" statement also clears the input buffer by prompting the '\n' to the nextline literal but generally it is avoided as it is only defined for the C++ versions below 11 standards.

---

## C++

```
// C++ Code to explain working of "flush(stdin);"
// discards the input buffer
#include <cstdio> //fflush(stdin) is available in cstdio header files
#include <iostream>
#include <iostream>
using namespace std;

int main()
{
    int value;
    string letters;

    // Enter an integer input from user
    // 98 for example
    cin >> value;

    // Discards the input buffer
    fflush(stdin);

    // Get input from user -
    // GeeksforGeeks for example
    getline(cin, letters);

    // Prints 98
    cout << value << endl;

    // Printing user entered string - this
    // will print string now
    cout << letters << endl;

    return 0;
}
```

```
// This code is contributed by im_impossible_ksv
```

Input:

```
369  
geeksforgeeks
```

Output:

```
369  
geeksforgeeks
```

### Time Complexity: O(1)

This article is contributed by **Manjeet Singh**. If you like GeeksforGeeks and would like to contribute, you can also write an article and mail your article to review-team@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

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

290

## Related Articles

1. [main Function in C](#)

---
2. [printf in C](#)

---
3. [C Program to Implement Max Heap](#)

---
4. [C Program to Implement Min Heap](#)

---
5. [C++ Error - Does not name a type](#)

---
6. [Execution Policy of STL Algorithms in Modern C++](#)

---
7. [C++ Program To Print Pyramid Patterns](#)

---
8. [Jagged Arrays in C++](#)

---

SALE!  
GeeksforGeeks Courses Upto 25% Off Enroll Now!



Save 25% on Courses DSA Data Structures Algorithms Interview Preparation Data Science T

# Operators in C

Difficulty Level : Easy • Last Updated : 04 Apr, 2023

[Read](#) [Discuss](#) [Courses](#) [Practice](#) [Video](#)

C Operators are symbols that represent operations to be performed on one or more operands. C provides a wide range of operators, which can be classified into different categories based on their functionality. Operators are used for performing operations on variables and values.

## What are Operators in C?

Operators can be defined as the symbols that help us to perform specific mathematical, relational, bitwise, conditional, or logical computations on operands. In other words, we can say that an operator operates the operands. For example, '+' is an operator used for addition, as shown below:

```
c = a + b;
```

Here, '+' is the operator known as the addition operator, and 'a' and 'b' are operands. The addition operator tells the compiler to add both of the operands 'a' and 'b'. The functionality of the C programming language is incomplete without the use of operators.

## Types of Operators in C

C has many built-in operators and can be classified into 6 types:

AD

1. Arithmetic Operators
2. Relational Operators
3. Logical Operators
4. Bitwise Operators
5. Assignment Operators
6. Other Operators

# Operators in C

	Operators	Type
Unary Operator	<code>++, --</code>	Unary Operator
Binary Operator	<code>+, -, *, /, %</code>	Arithmetic Operator
	<code>&lt;, &lt;=, &gt;, &gt;=, ==, !=</code>	Relational Operator
	<code>&amp;&amp;,   , !</code>	Logical Operator
	<code>&amp;,  , &lt;&lt;, &gt;&gt;, ~, ^</code>	Bitwise Operator
Ternary Operator	<code>=, +=, -=, *=, /=, %=</code>	Assignment Operator
	<code>?:</code>	Ternary or Conditional Operator

*Operators in C*

The above operators have been discussed in detail:

## 1. Arithmetic Operations in C

These operators are used to perform arithmetic/mathematical operations on operands.

Examples: `(+, -, *, /, %, ++, -)`. Arithmetic operators are of two types:

### a) Unary Operators:

Operators that operate or work with a single operand are unary operators. For example: Increment(`++`) and Decrement(`-`) Operators

```
int val = 5;
cout<<++val; // 6
```

### b) Binary Operators:

Operators that operate or work with two operands are binary operators. For example: Addition(+), Subtraction(-), multiplication(\*), Division(/) operators

```
int a = 7;
int b = 2;
cout<<a+b; // 9
```

## 2. Relational Operators in C

These are used for the comparison of the values of two operands. For example, checking if one operand is equal to the other operand or not, whether an operand is greater than the other operand or not, etc. Some of the relational operators are (==, >=, <=) (See [this](#) article for more reference).

```
int a = 3;
int b = 5;
cout<<(a < b);
// operator to check if a is smaller than b
```

## 3. Logical Operator in C

Logical Operators are used to combining two or more conditions/constraints or to complement the evaluation of the original condition in consideration. The result of the operation of a logical operator is a Boolean value either **true** or **false**.

For example, the **logical AND** represented as the '**&&**' operator in C returns true when both the conditions under consideration are satisfied. Otherwise, it returns false. Therefore, a && b returns true when both a and b are true (i.e. non-zero) (See [this](#) article for more reference).

```
cout<<((4 != 5) && (4 < 5)); // true
```

## 4. Bitwise Operators in C

The Bitwise operators are used to perform bit-level operations on the operands. The operators are first converted to bit-level and then the calculation is performed on the operands. Mathematical operations such as addition, subtraction, multiplication, etc. can

be performed at the bit level for faster processing. For example, the **bitwise AND** operator represented as '**&**' in C takes two numbers as operands and does AND on every bit of two numbers. The result of AND is 1 only if both bits are 1 (True).

```
int a = 5, b = 9;    // a = 5(00000101), b = 9(00001001)
cout << (a ^ b);   // 00001100
cout << (~a);      // 11111010
```

## 5. Assignment Operators in C

Assignment operators are used to assign value to a variable. The left side operand of the assignment operator is a variable and the right side operand of the assignment operator is a value. The value on the right side must be of the same data type as the variable on the left side otherwise the compiler will raise an error.

Different types of assignment operators are shown below:

### a) "="

This is the simplest assignment operator. This operator is used to assign the value on the right to the variable on the left.

#### Example:

```
a = 10;
b = 20;
ch = 'y';
```

### b) "+="

This operator is the combination of the '+' and '=' operators. This operator first adds the current value of the variable on left to the value on the right and then assigns the result to the variable on the left.

#### Example:

`(a += b)` can be written as `(a = a + b)`  
 If initially value stored in `a` is 5. Then `(a += 6) = 11.`

### c) "-="

This operator is a combination of '-' and '=' operators. This operator first subtracts the value on the right from the current value of the variable on left and then assigns the result to the

variable on the left.

#### **Example:**

(`a -= b`) can be written as (`a = a - b`)

If initially value stored in `a` is 8. Then (`a -= 6`) = 2.

#### **d) "\*="**

This operator is a combination of the '\*' and '=' operators. This operator first multiplies the current value of the variable on left to the value on the right and then assigns the result to the variable on the left.

#### **Example:**

(`a *= b`) can be written as (`a = a * b`)

If initially, the value stored in `a` is 5. Then (`a *= 6`) = 30.

#### **e) "/="**

This operator is a combination of the '/' and '=' operators. This operator first divides the current value of the variable on left by the value on the right and then assigns the result to the variable on the left.

#### **Example:**

(`a /= b`) can be written as (`a = a / b`)

If initially, the value stored in `a` is 6. Then (`a /= 2`) = 3.

## **6. Other Operators**

Apart from the above operators, there are some other operators available in C used to perform some specific tasks. Some of them are discussed here:

### **i. sizeof operator**

- `sizeof` is much used in the C programming language.
- It is a compile-time unary operator which can be used to compute the size of its operand.
- The result of `sizeof` is of the unsigned integral type which is usually denoted by `size_t`.
- Basically, the `sizeof` operator is used to compute the size of the variable.

To know more about the topic refer to [this](#) article.

## ii. Comma Operator

- The comma operator (represented by the token) is a binary operator that evaluates its first operand and discards the result, it then evaluates the second operand and returns this value (and type).
- The comma operator has the lowest precedence of any C operator.
- Comma acts as both operator and separator.

To know more about the topic refer to [this](#) article.

## iii. Conditional Operator

- The conditional operator is of the form ***Expression1? Expression2: Expression3***
- Here, Expression1 is the condition to be evaluated. If the condition(Expression1) is *True* then we will execute and return the result of Expression2 otherwise if the condition(Expression1) is *false* then we will execute and return the result of Expression3.
- We may replace the use of if..else statements with conditional operators.

To know more about the topic refer to [this](#) article.

## iv. dot (.) and arrow (->) Operators

- Member operators are used to referencing individual members of classes, structures, and unions.
- The dot operator is applied to the actual object.
- The arrow operator is used with a pointer to an object.

To know more about dot operators refer to [this](#) article and to know more about arrow(->) operators refer to [this](#) article.

## v. Cast Operator

- Casting operators convert one data type to another. For example, int(2.2000) would return 2.
- A cast is a special operator that forces one data type to be converted into another.
- The most general cast supported by most of the C compilers is as follows – **[ (type) expression ]**.

To know more about the topic refer to [this](#) article.

## vi. &, \* Operator

- Pointer operator & returns the address of a variable. For example &a; will give the actual address of the variable.
- The pointer operator \* is a pointer to a variable. For example \*var; will point to a variable var.

To know more about the topic refer to [this](#) article.

# C Operators with Example

---

## C

```
// C Program to Demonstrate the working concept of
// Operators
#include <stdio.h>

int main()
{
    int a = 10, b = 5;
    // Arithmetic operators
    printf("Following are the Arithmetic operators in C\n");
    printf("The value of a + b is %d\n", a + b);
    printf("The value of a - b is %d\n", a - b);

    printf("The value of a * b is %d\n", a * b);
    printf("The value of a / b is %d\n", a / b);
    printf("The value of a % b is %d\n", a % b);
    // First print (a) and then increment it
    // by 1
    printf("The value of a++ is %d\n", a++);

    // First print (a+1) and then decrease it
    // by 1
    printf("The value of a-- is %d\n", a--);

    // Increment (a) by (a+1) and then print
    printf("The value of ++a is %d\n", ++a);

    // Decrement (a+1) by (a) and then print
    printf("The value of --a is %d\n", --a);

    // Assignment Operators --> used to assign values to
    // variables int a =3, b=9; char d='d';

    // Comparison operators
    // Output of all these comparison operators will be (1)
    // if it is true and (0) if it is false
    printf(
        "\nFollowing are the comparison operators in C\n");
    printf("The value of a == b is %d\n", (a == b));
}
```

```

printf("The value of a != b is %d\n", (a != b));
printf("The value of a >= b is %d\n", (a >= b));
printf("The value of a <= b is %d\n", (a <= b));
printf("The value of a > b is %d\n", (a > b));
printf("The value of a < b is %d\n", (a < b));

// Logical operators
printf("\nFollowing are the logical operators in C\n");
printf("The value of this logical and operator ((a==b) && (a<b)) is:%d\n",
       ((a == b) && (a < b)));
printf("The value of this logical or operator ((a==b) || (a < b)) is:%d\n",
       ((a == b) || (a < b)));
printf("The value of this logical not operator !(a==b) is:%d\n",
       (!(a == b)));
}

return 0;
}

```

## Output

Following are the Arithmetic operators in C

The value of a + b is 15  
 The value of a - b is 5  
 The value of a \* b is 50  
 The value of a / b is 2  
 The value of a % b is 0  
 The value of a++ is 10  
 The value of a-- is 11  
 The value of ++a is 11  
 The value of --a is 10

Following are the comparison operators in C

The value of a == b is 0  
 The value of a != b is 1  
 The value of a >= b is 1  
 The value of a <= b is 0  
 The value of a > b is 1  
 The value of a < b is 0

Following are the logical operators in C

The value of this logical and operator ((a==b) && (a<b)) is:0  
 The value of this logical or operator ((a==b) || (a < b)) is:0  
 The value of this logical not operator !(a==b) is:1

## Time and Space Complexity

**Time Complexity:** O(1)

**Auxiliary Space:** O(1)

## Precedence of Operators in C

The below table describes the precedence order and associativity of operators in C. The precedence of the operator decreases from top to bottom.

Precedence	Operator	Description	Associativity
1	( )	Parentheses (function call)	left-to-right
	[ ]	Brackets (array subscript)	left-to-right
	.	Member selection via object name	left-to-right
	->	Member selection via a pointer	left-to-right
	a++/a-	Postfix increment/decrement (a is a variable)	left-to-right
2	++a/-a	Prefix increment/decrement (a is a variable)	right-to-left
	+/-	Unary plus/minus	right-to-left
	!~	Logical negation/bitwise complement	right-to-left
	(type)	Cast (convert value to temporary value of type)	right-to-left
	*	Dereference	right-to-left
	&	Address (of operand)	right-to-left

Precedence	Operator	Description	Associativity
	<b>sizeof</b>	Determine size in bytes on this implementation	right-to-left
3	<b>* , / , %</b>	Multiplication/division/modulus	left-to-right
4	<b>+/-</b>	Addition/subtraction	left-to-right
5	<b>&lt;&lt; , &gt;&gt;</b>	Bitwise shift left, Bitwise shift right	left-to-right
	<b>&lt; , &lt;=</b>	Relational less than/less than or equal to	left-to-right
6	<b>&gt; , &gt;=</b>	Relational greater than/greater than or equal to	left-to-right
7	<b>== , !=</b>	Relational is equal to/is not equal to	left-to-right
8	<b>&amp;</b>	Bitwise AND	left-to-right
9	<b>^</b>	Bitwise exclusive OR	left-to-right
10	<b> </b>	Bitwise inclusive OR	left-to-right
11	<b>&amp;&amp;</b>	Logical AND	left-to-right
12	<b>  </b>	Logical OR	left-to-right
13	<b>?:</b>	Ternary conditional	right-to-left
14	<b>=</b>	Assignment	right-to-left

Precedence	Operator	Description	Associativity
	<code>+= , -=</code>	Addition/subtraction assignment	right-to-left
	<code>*= , /=</code>	Multiplication/division assignment	right-to-left
	<code>%= , &amp;=</code>	Modulus/bitwise AND assignment	right-to-left
	<code>^= ,  =</code>	Bitwise exclusive/inclusive OR assignment	right-to-left
	<code>&lt;&gt;=</code>	Bitwise shift left/right assignment	right-to-left
15	,	expression separator	left-to-right

## Conclusion

In this article, the points we learned about the operator are as follows:

- Operators are symbols used for performing some kind of operation in C.
- The operation can be mathematical, logical, relational, bitwise, conditional, or logical.
- There are seven types of Unary operators, Arithmetic operator, Relational operator, Logical operator, Bitwise operator, Assignment operator, and Conditional operator.
- Every operator returns a numerical value except logical and conditional operator which returns a boolean value (true or false).
- '=' and '==' are not same as '=' assigns the value whereas '==' checks if both the values are equal or not.
- There is a Precedence in the operator means the priority of using one operator is greater than another operator.

## Frequently Asked Questions(FAQs)

### 1. What are operators in C?

Operators in C are certain symbols in C used for performing certain mathematical, relational, bitwise, conditional, or logical operations for the user.

## 2. What are the 7 types of operators in C?

There are 7 types of operators in C as mentioned below:

- Unary operator
- Arithmetic operator
- Relational operator
- Logical operator
- Bitwise operator
- Assignment operator
- Conditional operator

## 3. What is the difference between the '=' and '==' operators?

'=' is a type of assignment operator that places the value in right to the variable on left, Whereas '==' is a type of relational operator that is used to compare two elements if the elements are equal or not.

## 4. What is the difference between prefix and postfix operators in C?

Prefix operations are the operations in which the value is returned prior to the operation whereas in postfix operations value is returned after updating the value in the variable.

### Example:

```
b=c=10;  
a=b++; // a==10  
a=++c; // a==11
```

## 5. What is the Modulo operator?

The Modulo operator(%) is used to find the remainder if one element is divided by another.

### Example:

```
a % b (a divided by b)  
5 % 2 == 1
```

**SALE!**  
**GeeksforGeeks Courses Upto 25% Off Enroll Now!**

[Save 25% on Courses](#)[DSA](#)[Data Structures](#)[Algorithms](#)[Interview Preparation](#)[Data Science](#)[T](#)

# Unary operators in C/C++

Difficulty Level : Basic • Last Updated : 03 Apr, 2023

[Read](#)[Discuss](#)[Courses](#)[Practice](#)[Video](#)

**Unary operators:** are operators that act upon a single operand to produce a new value.

## Types of unary operators:

1. unary minus(-)
2. increment(++)
3. decrement(--)
4. NOT(!)
5. Addressof operator(&)
6. sizeof()

Time complexity of any unary operator is O(1).

Auxiliary Space of any unary operator is O(1).

AD

**1. unary minus:** The minus operator changes the sign of its argument. A positive number becomes negative, and a negative number becomes positive.

```
int a = 10;  
int b = -a; // b = -10
```

unary minus is different from the subtraction operator, as subtraction requires two operands.

Below is the implementation of **unary minus (-)** operator:

## C++

```
// C++ program to demonstrate the use of 'unary minus'
// operator

#include <iostream>
using namespace std;

int main()
{
    int positiveInteger = 100;
    int negativeInteger = -positiveInteger;

    cout << "Positive Integer: " << positiveInteger << endl;
    cout << "Negative Integer: " << negativeInteger << endl;

    return 0;
}

// This code is contributed by sarajadhav12052009
```

## Output

```
Positive Integer: 100
Negative Integer: -100
```

**2. increment:** It is used to increment the value of the variable by 1. The increment can be done in two ways:

**2.1 prefix increment:** In this method, the operator precedes the operand (e.g., `++a`). The value of the operand will be altered *before* it is used.

```
int a = 1;
int b = ++a; // b = 2
```

**2.2 postfix increment:** In this method, the operator follows the operand (e.g., `a++`). The value operand will be altered *after* it is used.

```
int a = 1;
int b = a++; // b = 1
int c = a; // c = 2
```

**3. decrement:** It is used to decrement the value of the variable by 1. The decrement can be done in two ways:

**3.1 prefix decrement:** In this method, the operator precedes the operand (e.g., `--a`). The value of the operand will be altered *before* it is used.

```
int a = 1;
int b = --a; // b = 0
```

**3.2 postfix decrement:** In this method, the operator follows the operand (e.g., `a--`). The value of the operand will be altered *after* it is used.

```
int a = 1;
int b = a--; // b = 1
int c = a; // c = 0
```

**4. NOT(!):** It is used to reverse the logical state of its operand. If a condition is true, then the Logical NOT operator will make it false.

If `x` is true, then `!x` is false  
 If `x` is false, then `!x` is true

Below is the implementation of the **NOT (!)** operator:

## C++

```
// C++ program to demonstrate the use of '! (NOT) operator'

#include <iostream>
using namespace std;

int main()
{
    int a = 10;
    int b = 5;

    if (!(a > b))
        cout << "b is greater than a" << endl;
    else
        cout << "a is greater than b" << endl;

    return 0;
}

// This code is contributed by sarajadhav12052009
```

## Output

```
a is greater than b
```

**5. Addressof operator(&):** It gives an address of a variable. It is used to return the memory address of a variable. These addresses returned by the address-of operator are known as pointers because they “point” to the variable in memory.

```
& gives an address on variable n
int a;
int *ptr;
ptr = &a; // address of a is copied to the location ptr.
```

Below is the implementation of **Addressof operator(&)**:

## C++

```
// C++ program to demonstrate the use of 'address-of(&)'
// operator

#include <iostream>
using namespace std;

int main()
{
    int a;
    int* ptr;

    ptr = &a;

    cout << ptr;

    return 0;
}

// This code is contributed by sarajadhav12052009
```

## Output

```
0x7ffddcf0c8ec
```

**6. sizeof():** This operator returns the size of its operand, in bytes. The `sizeof()` operator always precedes its operand. The operand is an expression, or it may be a cast.

**Note:** The ``sizeof()`` operator in C++ is machine dependent. For example, the size of an ‘int’ in C++ may be 4 bytes in a 32-bit machine but it may be 8 bytes in a 64-bit machine.

Below is the implementation of **sizeof()** operator:

## C++

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

int main()
{
    float n = 0;
    cout << "size of n: " << sizeof(n);
    return 0;
}
```

## Output

size of n: 4

167

## Related Articles

1. [Operators in C | Set 2 \(Relational and Logical Operators\)](#)
2. [What are the Operators that Can be and Cannot be Overloaded in C++?](#)
3. [Increment \(Decrement\) operators require L-value Expression](#)
4. [Order of operands for logical operators](#)
5. [Conversion Operators in C++](#)
6. [const\\_cast in C++ | Type Casting operators](#)
7. [How to sum two integers without using arithmetic operators in C/C++?](#)
8. [Execution of printf With ++ Operators in C](#)
9. [Conditionally assign a value without using conditional and arithmetic operators](#)
10. [new and delete Operators in C++ For Dynamic Memory](#)

**SALE!****GeeksforGeeks Courses Upto 25% Off Enroll Now!****Save 25% on Courses**

DSA

Data Structures

Algorithms

Interview Preparation

Data Science

T

# Pre-increment (or pre-decrement) With Reference to L-value in C++

Difficulty Level : Medium • Last Updated : 22 Jun, 2022

**Read**

Discuss(50+)

Courses

Practice

Video

## Prerequisite: Pre-increment and post-increment in C/C++

In C++, pre-increment (or pre-decrement) can be used as l-value, but post-increment (or post-decrement) can not be used as l-value.

For example, following program prints  $a = 20$  ( $++a$  is used as l-value)

l-value is simply nothing but the memory location, which has an address.

## CPP

```
// CPP program to illustrate
// Pre-increment (or pre-decrement)
#include <cstdio>

int main()
{
    int a = 10;

    ++a = 20; // works
    printf("a = %d", a);
    printf("\n");
    --a = 10;
    printf("a = %d", a);
    return 0;
}
```

## Output:

AD

```
a = 20
a = 10
```

### Time Complexity: O(1)

The above program works whereas the following program fails in compilation with error “*non-lvalue in assignment*” (`a++` is used as l-value)

## CPP

```
// CPP program to illustrate
// Post-increment (or post-decrement)
#include <cstdio>

int main()
{
    int a = 10;
    a++ = 20; // error
    printf("a = %d", a);
    return 0;
}
```

### Error:

```
prog.cpp: In function 'int main()':
prog.cpp:6:5: error: lvalue required as left operand of assignment
    a++ = 20; // error
           ^
```

### How `++a` is Different From `a++` as lvalue?

It is because `++a` returns an *lvalue*, which is basically a reference to the variable to which we can further assign – just like an ordinary variable. It could also be assigned to a reference as follows:

```
int &ref = ++a; // valid
int &ref = a++; // invalid
```

Whereas if you recall how `a++` works, it doesn't immediately increment the value it holds. For clarity, you can think of it as getting incremented in the next statement. So what basically happens is that, `a++` returns an *rvalue*, which is basically just a value like the value of an expression that is not stored. You can think of `a++ = 20;` as follows after being processed:

```
int a = 10;

// On compilation, a++ is replaced by the value of a which is an rvalue:
10 = 20; // Invalid

// Value of a is incremented
a = a + 1;
```

That should help to understand why `a++ = 20;` won't work. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

144

## Related Articles

1. [lvalue and rvalue in C language](#)

---
2. [Output of the program | Dereference, Reference, Dereference, Reference....](#)

---
3. [When do we pass arguments by reference or pointer?](#)

---
4. [Reference to a pointer in C++ with examples and applications](#)

---
5. [Passing Reference to a Pointer in C++](#)

---
6. [Difference between Call by Value and Call by Reference](#)

---
7. [Can C++ reference member be declared without being initialized with declaration?](#)

---
8. [Different ways to use Const with Reference to a Pointer in C++](#)

---

**SALE!**  
**GeeksforGeeks Courses Upto 25% Off Enroll Now!**



Save 25% on Courses DSA Data Structures Algorithms Interview Preparation Data Science T

# new and delete Operators in C++ For Dynamic Memory

Difficulty Level : Easy • Last Updated : 18 Oct, 2022

Read

Discuss(20+)

Courses

Practice

Video

Dynamic memory allocation in C/C++ refers to performing memory allocation manually by a programmer. Dynamically allocated memory is allocated on **Heap**, and non-static and local variables get memory allocated on **Stack** (Refer to [Memory Layout C Programs](#) for details).

## What are applications?

- One use of dynamically allocated memory is to allocate memory of variable size, which is not possible with compiler allocated memory except for [variable-length arrays](#).
- The most important use is the flexibility provided to programmers. We are free to allocate and deallocate memory whenever we need it and whenever we don't need it anymore. There are many cases where this flexibility helps. Examples of such cases are [Linked List](#), [Tree](#), etc.

## How is it different from memory allocated to normal variables?

For normal variables like "int a", "char str[10]", etc, memory is automatically allocated and deallocated. For dynamically allocated memory like "int \*p = new int[10]", it is the programmer's responsibility to deallocate memory when no longer needed. If the programmer doesn't deallocate memory, it causes a [memory leak](#) (memory is not deallocated until the program terminates).

## How is memory allocated/deallocated in C++?

C uses the [malloc\(\) and calloc\(\)](#) function to allocate memory dynamically at run time and uses a [free\(\)](#) function to free dynamically allocated memory. C++ supports these functions

and also has two operators **new** and **delete**, that perform the task of allocating and freeing the memory in a better and easier way.

AD

## **new operator**

The new operator denotes a request for memory allocation on the Free Store. If sufficient memory is available, a new operator initializes the memory and returns the address of the newly allocated and initialized memory to the pointer variable.

### **Syntax to use new operator**

```
pointer-variable = new data-type;
```

Here, the pointer variable is the pointer of type data-type. Data type could be any built-in data type including array or any user-defined data type including structure and class.

#### **Example:**

```
// Pointer initialized with NULL
// Then request memory for the variable
int *p = NULL;
p = new int;
```

OR

```
// Combine declaration of pointer
// and their assignment
int *p = new int;
```

**Initialize memory:** We can also initialize the memory for built-in data types using a new operator. For custom data types, a constructor is required (with the data type as input) for initializing the value. Here's an example of the initialization of both data types :

```
pointer-variable = new data-type(value);
```

**Example:****C++**

```

int* p = new int(25);
float* q = new float(75.25);

// Custom data type
struct cust
{
    int p;
    cust(int q) : p(q) {}
    cust() = default;
    //cust& operator=(const cust& that) = default;
};

int main()
{
    // Works fine, doesn't require constructor
    cust* var1 = new cust;

    //OR

    // Works fine, doesn't require constructor
    var1 = new cust();

    // Notice error if you comment this line
    cust* var = new cust(25);
    return 0;
}

```

**Allocate a block of memory:** a new operator is also used to allocate a block(an array) of memory of type *data type*.

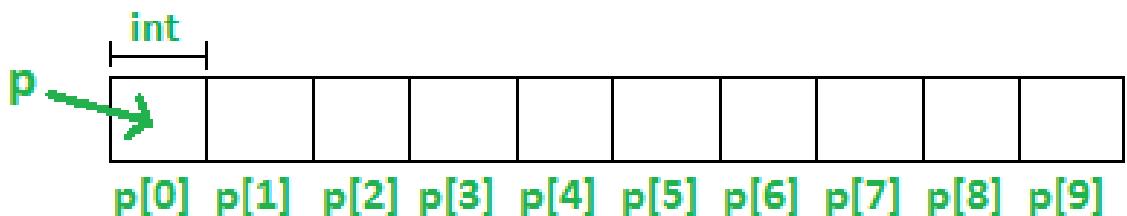
```
pointer-variable = new data-type[size];
```

where size(a variable) specifies the number of elements in an array.

**Example:**

```
int *p = new int[10]
```

Dynamically allocates memory for 10 integers continuously of type int and returns a pointer to the first element of the sequence, which is assigned to p(a pointer). p[0] refers to the first element, p[1] refers to the second element, and so on.



### Normal Array Declaration vs Using new

There is a difference between declaring a normal array and allocating a block of memory using new. The most important difference is, that normal arrays are deallocated by the compiler (If the array is local, then deallocated when the function returns or completes). However, dynamically allocated arrays always remain there until either they are deallocated by the programmer or the program terminates.

### What if enough memory is not available during runtime?

If enough memory is not available in the heap to allocate, the new request indicates failure by throwing an exception of type std::bad\_alloc, unless "nothrow" is used with the new operator, in which case it returns a NULL pointer (scroll to section "Exception handling of new operator" in [this article](#)). Therefore, it may be a good idea to check for the pointer variable produced by the new before using its program.

```
int *p = new(nothrow) int;
if (!p)
{
    cout << "Memory allocation failed\n";
}
```

### delete operator

Since it is the programmer's responsibility to deallocate dynamically allocated memory, programmers are provided delete operator in C++ language.

#### Syntax:

```
// Release memory pointed by pointer-variable
delete pointer-variable;
```

Here, the pointer variable is the pointer that points to the data object created by ***new***.

### Examples:

```
delete p;
delete q;
```

To free the dynamically allocated array pointed by pointer variable, use the following form of *delete*:

```
// Release block of memory
// pointed by pointer-variable
delete[] pointer-variable;
```

### Example:

```
// It will free the entire array
// pointed by p.
delete[] p;
```

## CPP

```
// C++ program to illustrate dynamic allocation
// and deallocation of memory using new and delete
#include <iostream>
using namespace std;

int main ()
{
    // Pointer initialization to null
    int* p = NULL;

    // Request memory for the variable
    // using new operator
    p = new(nothrow) int;
    if (!p)
        cout << "allocation of memory failed\n";
    else
    {
        // Store value at allocated address
        *p = 29;
        cout << "Value of p: " << *p << endl;
    }

    // Request block of memory
    // using new operator
    float *r = new float(75.25);

    cout << "Value of r: " << *r << endl;
}
```

```

// Request block of memory of size n
int n = 5;
int *q = new(nothrow) int[n];

if (!q)
    cout << "allocation of memory failed\n";
else
{
    for (int i = 0; i < n; i++)
        q[i] = i+1;

    cout << "Value store in block of memory: ";
    for (int i = 0; i < n; i++)
        cout << q[i] << " ";
}

// freed the allocated memory
delete p;
delete r;

// freed the block of allocated memory
delete[] q;

return 0;
}

```

## Output

```

Value of p: 29
Value of r: 75.25
Value store in block of memory: 1 2 3 4 5

```

**Time Complexity:** O(n), where n is the given memory size.

### Related Articles:

- [Quiz on new and delete](#)
- [delete vs free](#)

This article is contributed by **Akash Gupta**. If you like GeeksforGeeks and would like to contribute, you can also write an article using [write.geeksforgeeks.org](https://write.geeksforgeeks.org) or mail your article to [review-team@geeksforgeeks.org](mailto:review-team@geeksforgeeks.org). See your article appearing on the GeeksforGeeks main page and help other Geeks. Please write comments if you find anything incorrect, or if you want to share more information about the topic discussed above.

## Related Articles

SALE!  
GeeksforGeeks Courses Upto 25% Off Enroll Now!



Save 25% on Courses DSA Data Structures Algorithms Array Strings Linked List Stack Q

# C Arrays

Difficulty Level : Easy • Last Updated : 06 Apr, 2023

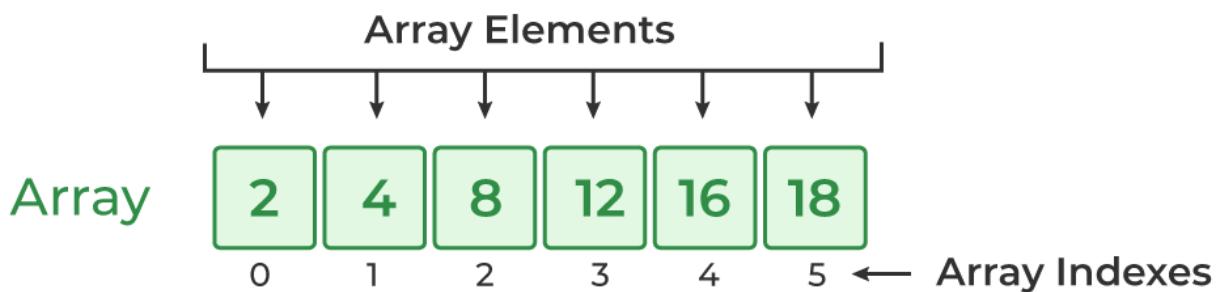
[Read](#) [Discuss\(30+\)](#) [Courses](#) [Practice](#) [Video](#)

**Array in C** is one of the most used data structures in C programming. It is a simple and fast way of storing multiple values under a single name. In this article, we will study the different aspects of array in C language such as array declaration, definition, initialization, types of arrays, array syntax, advantages and disadvantages, and many more.

## What is Array in C?

An array in C is a fixed-size collection of similar data items stored in contiguous memory locations. It can be used to store the collection of primitive data types such as int, char, float, etc., and also derived and user-defined data types such as pointers, structures, etc.

## Array in C



## C Array Declaration

In C, we have to declare the array like any other variable before using it. We can declare an array by specifying its name, the type of its elements, and the size of its dimensions. When

we declare an array in C, the compiler allocates the memory block of the specified size to the array name.

## Syntax of Array Declaration

*data\_type array\_name [size];*

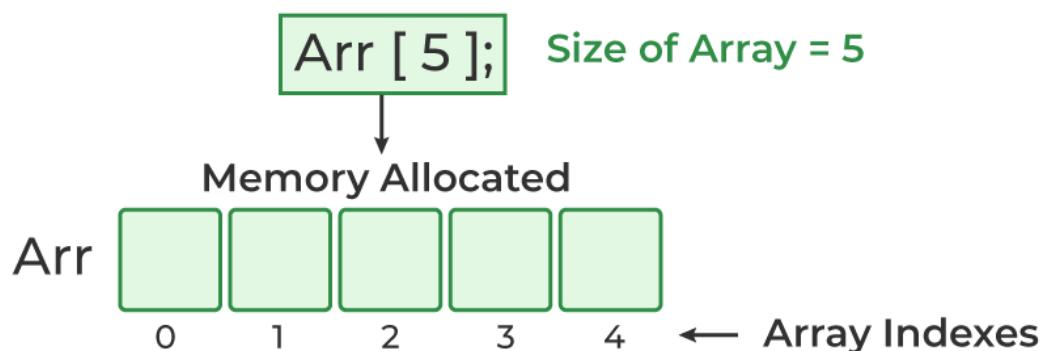
or

*data\_type array\_name [size1] [size2]...[sizeN];*

where N is the number of dimensions.

AD

## Array Declaration



The C arrays are static in nature, i.e., they are allocated memory at the compile time.

## Example of Array Declaration

### C

```
// C Program to illustrate the array declaration
#include <stdio.h>
```

```

int main()
{
    // declaring array of integers
    int arr_int[5];
    // declaring array of characters
    char arr_char[5];

    return 0;
}

```

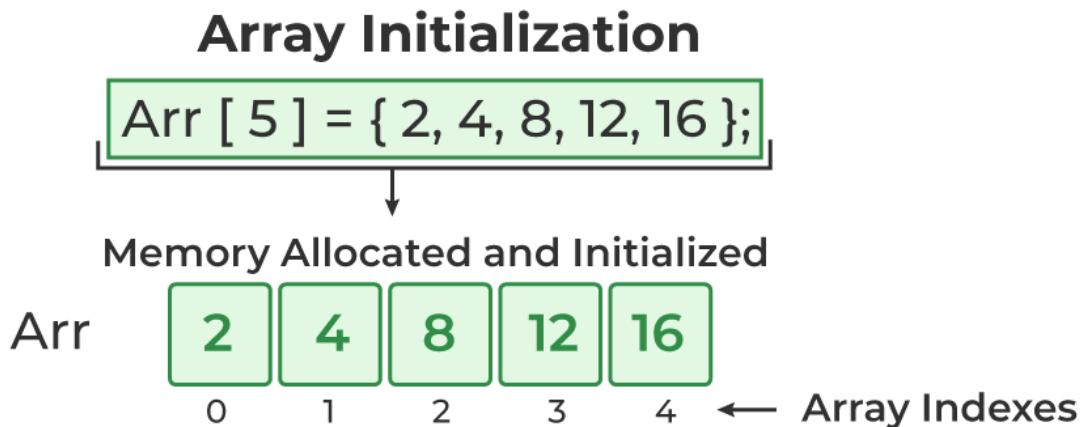
## C Array Initialization

Initialization in C is the process to assign some initial value to the variable. When the array is declared or allocated memory, the elements of the array contain some garbage value. So, we need to initialize the array to some meaningful value. There are multiple ways in which we can initialize an array in C.

### 1. Array Initialization with Declaration

In this method, we initialize the array along with its declaration. We use an initializer list to initialize multiple elements of the array. An initializer list is the list of values enclosed within braces {} separated by a comma.

```
data_type array_name [size] = {value1, value2, ... valueN};
```



### 2. Array Initialization with Declaration without Size

If we initialize an array using an initializer list, we can skip declaring the size of the array as the compiler can automatically deduce the size of the array in these cases. The size of the array in these cases is equal to the number of elements present in the initializer list as the compiler can automatically deduce the size of the array.

```
data_type array_name[] = {1,2,3,4,5};
```

The size of the above arrays is 5 which is automatically deduced by the compiler.

### 3. Array Initialization after Declaration (Using Loops)

We initialize the array after the declaration by assigning the initial value to each element individually. We can use for loop, while loop, or do-while loop to assign the value to each element of the array.

```
for (int i = 0; i < N; i++) {
    array_name[i] = valuei;
}
```

## Example of Array Initialization in C

---

### C

```
// C Program to demonstrate array initialization
#include <stdio.h>

int main()
{
    // array initialization using initializer list
    int arr[5] = { 10, 20, 30, 40, 50 };

    // array initialization using initializer list without
    // specifying size
    int arr1[] = { 1, 2, 3, 4, 5 };

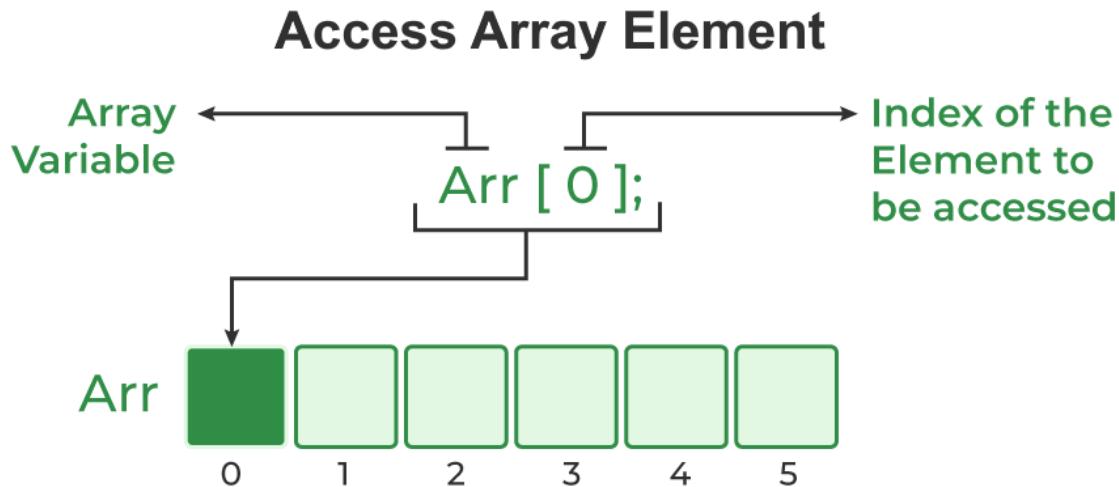
    // array initialization using for loop
    float arr2[5];
    for (int i = 0; i < 5; i++) {
        arr2[i] = (float)i * 2.1;
    }
    return 0;
}
```

## Access Array Elements

We can access any element of an array in C using the array subscript operator `[ ]` and the index value *i* of the element.

```
array_name [index];
```

One thing to note is that the indexing in the array always starts with 0, i.e., the **first element** is at index **0** and the **last element** is at **N - 1** where **N** is the number of elements in the array.



### Example of Accessing Array Elements using Array Subscript Operator

#### C

```
// C Program to illustrate element access using array
// subscript
#include <stdio.h>

int main()
{
    // array declaration and initialization
    int arr[5] = { 15, 25, 35, 45, 55 };

    // accessing element at index 2 i.e 3rd element
    printf("Element at arr[2]: %d\n", arr[2]);

    // accessing element at index 4 i.e last element
    printf("Element at arr[4]: %d\n", arr[4]);

    // accessing element at index 0 i.e first element
    printf("Element at arr[0]: %d", arr[0]);
```

```
    return 0;  
}
```

## Output

```
Element at arr[2]: 35  
Element at arr[4]: 55  
Element at arr[0]: 15
```

## Update Array Element

We can update the value of an element at the given index  $i$  in a similar way to accessing an element by using the array subscript operator `[]` and assignment operator `=`.

```
array_name[i] = new_value;
```

## C Array Traversal

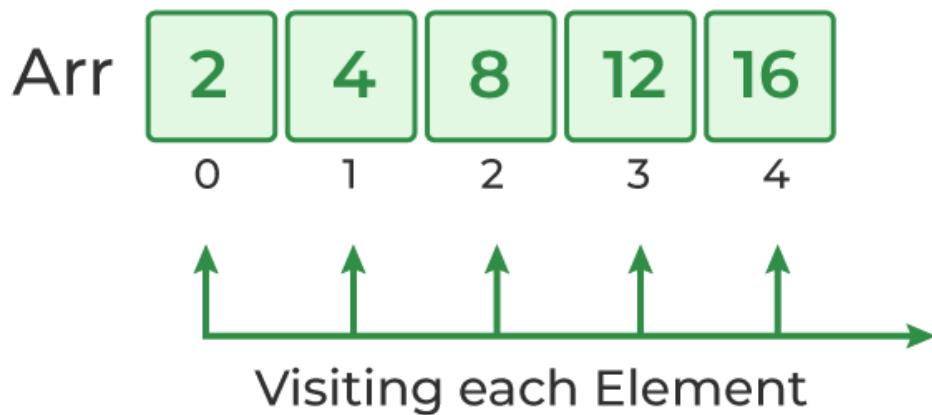
Traversal is the process in which we visit every element of the data structure. For C array traversal, we use loops to iterate through each element of the array.

### Array Traversal using for Loop

```
for (int i = 0; i < N; i++) {  
    array_name[i];  
}
```

# Array Transversal

```
for ( int i = 0; i < Size; i++ ){
    arr[i];
}
```



## How to use Array in C?

The following program demonstrates how to use an array in the C programming language:

### C

```
// C Program to demonstrate the use of array
#include <stdio.h>

int main()
{
    // array declaration and initialization
    int arr[5] = { 10, 20, 30, 40, 50 };

    // modifying element at index 2
    arr[2] = 100;

    // traversing array using for loop
    printf("Elements in Array: ");
    for (int i = 0; i < 5; i++) {
        printf("%d ", arr[i]);
    }

    return 0;
}
```

}

## Output

```
Elements in Array: 10 20 100 40 50
```

## Types of Array in C

There are two types of arrays based on the number of dimensions it has. They are as follows:

1. One Dimensional Arrays (1D Array)
2. Multidimensional Arrays

## 1. One Dimensional Array in C

The One-dimensional arrays, also known as 1-D arrays in C are those arrays that have only one dimension.

### Syntax of 1D Array in C

```
array_name [size];
```

## 1D Array

1	2	3	4	5	6	7	8
---	---	---	---	---	---	---	---

### Example of 1D Array in C

---

**C**

```
// C Program to illustrate the use of 1D array
#include <stdio.h>

int main()
{
    // 1d array declaration
    int arr[5];

    // 1d array initialization using for loop
    for (int i = 0; i < 5; i++) {
        arr[i] = i * i - 2 * i + 1;
    }

    printf("Elements of Array: ");
    // printing 1d array by traversing using for loop
    for (int i = 0; i < 5; i++) {
        printf("%d ", arr[i]);
    }

    return 0;
}
```

## Output

Elements of Array: 1 0 1 4 9

## Array of Characters (Strings)

In C, we store the words, i.e., a sequence of characters in the form of an array of characters terminated by a NULL character. These are called strings in C language.

---

## C

```
// C Program to illustrate strings
#include <stdio.h>

int main()
{
    // creating array of character
    char arr[6] = { 'G', 'e', 'e', 'k', 's', '\0' };

    // printing string
    int i = 0;
    while (arr[i]) {
        printf("%c", arr[i++]);
    }

    return 0;
}
```

## Output

Geeks

To know more about strings, refer to this article – [Strings in C](#)

## 2. Multidimensional Array in C

Multi-dimensional Arrays in C are those arrays that have more than one dimension. Some of the popular multidimensional arrays are 2D arrays and 3D arrays. We can declare arrays with more dimensions than 3d arrays but they are avoided as they get very complex and occupy a large amount of space.

### A. Two-Dimensional Array in C

A Two-Dimensional array or 2D array in C is an array that has exactly two dimensions. They can be visualized in the form of rows and columns organized in a two-dimensional plane.

#### Syntax of 2D Array in C

```
array_name[size1] [size2];
```

Here,

- **size1:** Size of the first dimension.
- **size2:** Size of the second dimension.

# 2D Array

1	2	3	4
1	2	3	4
1	2	3	4
1	2	3	4

## Example of 2D Array in C

C

```
// C Program to illustrate 2d array
#include <stdio.h>

int main()
{
    // declaring and initializing 2d array
    int arr[2][3] = { 10, 20, 30, 40, 50, 60 };

    printf("2D Array:\n");
    // printing 2d array
    for (int i = 0; i < 2; i++) {
        for (int j = 0; j < 3; j++) {
            printf("%d ",arr[i][j]);
        }
        printf("\n");
    }

    return 0;
}
```

## Output

2D Array:

```
10 20 30  
40 50 60
```

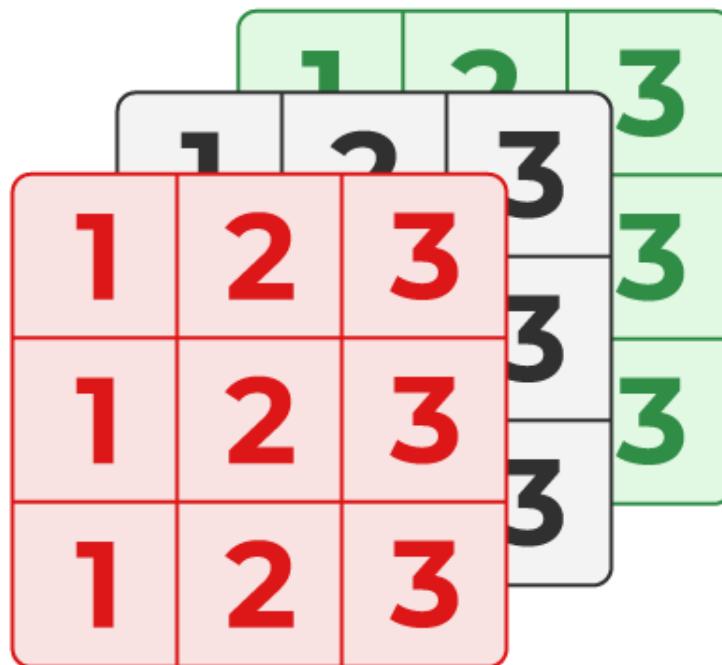
## B. Three-Dimensional Array in C

Another popular form of a multi-dimensional array is Three Dimensional Array or 3D Array. A 3D array has exactly three dimensions. It can be visualized as a collection of 2D arrays stacked on top of each other to create the third dimension.

### Syntax of 3D Array in C

```
array_name [size1] [size2] [size3];
```

# 3D Array



### Example of 3D Array

**C**

```
// C Program to illustrate the 3d array
#include <stdio.h>

int main()
{
    // 3D array declaration
    int arr[2][2][2] = { 10, 20, 30, 40, 50, 60 };

    // printing elements
    for (int i = 0; i < 2; i++) {
        for (int j = 0; j < 2; j++) {
            for (int k = 0; k < 2; k++) {
                printf("%d ", arr[i][j][k]);
            }
            printf("\n");
        }
        printf("\n \n");
    }
    return 0;
}
```

**Output**

10 20  
30 40

50 60  
0 0

To know more about Multidimensional Array in C, refer to this article – [Multidimensional Arrays in C](#)

**Relationship between Arrays and Pointers**

Arrays and Pointers are closely related to each other such that we can use pointers to perform all the possible operations of the array. The array name is a constant pointer to the first element of the array and the array decays to the pointers when passed to the function.

**C**

```
// C Program to demonstrate the relation between arrays and
```

```
// pointers
#include <stdio.h>

int main()
{
    int arr[5] = { 10, 20, 30, 40, 50 };
    int* ptr = &arr[0];

    // comparing address of first element and address stored
    // inside array name
    printf("Address Stored in Array name: %p\nAddress of "
        "1st Array Element: %p\n",
        arr, &arr[0]);

    // printing array elements using pointers
    printf("Array elements using pointer: ");
    for (int i = 0; i < 5; i++) {
        printf("%d ", *ptr++);
    }
    return 0;
}
```

## Output

```
Address Stored in Array name: 0x7ffce72c2660
Address of 1st Array Element: 0x7ffce72c2660
Array elements using pointer: 10 20 30 40 50
```

To know more about the relationship between an array and a pointer, refer to this article – [Pointer to an Arrays | Array Pointer](#)

## Passing an Array to a Function in C

An array is always passed as pointers to a function in C. Whenever we try to pass an array to a function, it decays to the pointer and then passed as a pointer to the first element of an array.

We can verify this using the following C Program:

## C

```
// C Program to pass an array to a function
#include <stdio.h>

void printArray(int arr[])
{
    printf("Size of Array in Functions: %d\n", sizeof(arr));
    printf("Array Elements: ");
    for (int i = 0; i < 5; i++) {
        printf("%d ", arr[i]);
    }
}
```

```

    }
}

// driver code
int main()
{
    int arr[5] = { 10, 20, 30, 40, 50 };

    printf("Size of Array in main(): %d\n", sizeof(arr));
    printArray(arr);
    return 0;
}

```

## Output

```

Size of Array in main(): 20
Size of Array in Functions: 8
Array Elements: 10 20 30 40 50

```

## Return an Array from a Function in C

In C, we can only return a single value from a function. To return multiple values or elements, we have to use pointers. We can return an array from a function using a pointer to the first element of that array.

---

## C

```

// C Program to return array from a function
#include <stdio.h>

// function
int* func()
{
    static int arr[5] = { 1, 2, 3, 4, 5 };

    return arr;
}

// driver code
int main()
{
    int* ptr = func();

    printf("Array Elements: ");
    for (int i = 0; i < 5; i++) {
        printf("%d ", *ptr++);
    }
    return 0;
}

```

## Output

```
Array Elements: 1 2 3 4 5
```

**Note:** You may have noticed that we declared static array using static keyword. This is due to the fact that when a function returns a value, all the local variables and other entities declared inside that function are deleted. So, if we create a local array instead of static, we will get segmentation fault while trying to access the array in the main function.

# Properties of Arrays in C

It is very important to understand the properties of the C array so that we can avoid bugs while using it. The following are the main [properties of an array in C](#):

## 1. Fixed Size

The array in C is a fixed-size collection of elements. The size of the array must be known at the compile time and it cannot be changed once it is declared.

## 2. Homogeneous Collection

We can only store one type of element in an array. There is no restriction on the number of elements but the type of all of these elements must be the same.

## 3. Indexing in Array

The array index always starts with 0 in C language. It means that the index of the first element of the array will be 0 and the last element will be  $N - 1$ .

## 4. Dimensions of an Array

A dimension of an array is the number of indexes required to refer to an element in the array. It is the number of directions in which you can grow the array size.

## 5. Contiguous Storage

All the elements in the array are stored continuously one after another in the memory. It is one of the defining properties of the array in C which is also the reason why random access is possible in the array.

## 6. Random Access

The array in C provides random access to its element i.e we can get to a random element at any index of the array in constant time complexity just by using its index number.

## 7. No Index Out of Bounds Checking

There is no index out-of-bounds checking in C/C++, for example, the following program compiles fine but may produce unexpected output when run.

### C

```
// This C program compiles fine
// as index out of bound
// is not checked in C.

#include <stdio.h>

int main()
{
    int arr[2];

    printf("%d ", arr[3]);
    printf("%d ", arr[-2]);

    return 0;
}
```

### Output

211343841 4195777

In C, it is not a compiler error to initialize an array with more elements than the specified size. For example, the below program compiles fine and shows just a Warning.

### C

```
#include <stdio.h>
int main()
{
    // Array declaration by initializing it
```

```
// with more elements than specified size.
int arr[2] = { 10, 20, 30, 40, 50 };

return 0;
}
```

## Warnings:

```
prog.c: In function 'main':
prog.c:7:25: warning: excess elements in array initializer
    int arr[2] = { 10, 20, 30, 40, 50 };
                           ^
prog.c:7:25: note: (near initialization for 'arr')
prog.c:7:29: warning: excess elements in array initializer
    int arr[2] = { 10, 20, 30, 40, 50 };
                           ^
prog.c:7:29: note: (near initialization for 'arr')
prog.c:7:33: warning: excess elements in array initializer
    int arr[2] = { 10, 20, 30, 40, 50 };
                           ^
prog.c:7:33: note: (near initialization for 'arr')
```

## Examples of Array in C

### Example 1: C Program to perform array input and output.

In this program, we will use `scanf()` and `printf()` function to take input and print output for the array.

## C

```
// C Program to perform input and output on array
#include <stdio.h>

int main()
{
    // declaring an integer array
    int arr[5];

    // taking input to array elements one by one
    for (int i = 0; i < 5; i++) {
        scanf("%d", &arr[i]);
    }

    // printing array elements
    printf("Array Elements: ");
}
```

```

for (int i = 0; i < 5; i++) {
    printf("%d ", arr[i]);
}
return 0;
}

```

**Input**

5 7 9 1 4

**Output**

Array Elements: 5 7 9 1 4

**Example 2: C Program to print the average of the given list of numbers**

In this program, we will store the numbers in an array and traverse it to calculate the average of the number stored.

---

**C**

```

// C Program to calculate average of two numbers
#include <stdio.h>

// function to calculate average of the function
float getAverage(float* arr, int size)
{
    int sum = 0;
    // calculating cumulative sum of all the array elements
    for (int i = 0; i < size; i++) {
        sum += arr[i];
    }

    // returning average
    return sum / size;
}

// driver code
int main()
{
    // declaring and initializing array
    float arr[5] = { 10, 20, 30, 40, 50 };
    // size of array using sizeof operator
    int n = sizeof(arr) / sizeof(float);

    // printing array elements
    printf("Array Elements: ");
    for (int i = 0; i < n; i++) {

```

```

    printf("%.0f ", arr[i]);
}

// calling getAverage function and printing average
printf("\nAverage: %.2f", getAverage(arr, n));
return 0;
}

```

## Output

Array Elements: 10 20 30 40 50  
 Average: 30.00

## Example 3: C Program to find the largest number in the array.

---

## C

```

// C Program to find the largest number in the array.
#include <stdio.h>

// function to return max value
int getMax(int* arr, int size)
{
    int max = arr[0];
    for (int i = 1; i < size; i++) {
        if (max < arr[i]) {
            max = arr[i];
        }
    }
    return max;
}

// Driver code
int main()
{
    int arr[10]
        = { 135, 165, 1, 16, 511, 65, 654, 654, 169, 4 };

    printf("Largest Number in the Array: %d",
        getMax(arr, 10));

    return 0;
}

```

## Output

Largest Number in the Array: 654

## Advantages of Array in C

The following are the main advantages of an array:

1. Random and fast access of elements using the array index.
2. Use of fewer lines of code as it creates a single array of multiple elements.
3. Traversal through the array becomes easy using a single loop.
4. Sorting becomes easy as it can be accomplished by writing fewer lines of code.

## Disadvantages of Array in C

1. Allows a fixed number of elements to be entered which is decided at the time of declaration. Unlike a linked list, an array in C is not dynamic.
2. Insertion and deletion of elements can be costly since the elements are needed to be rearranged after insertion and deletion.

## Conclusion

The array is one of the most used and important data structures in C. It is one of the core concepts of C language that is used in every other program. Though it is important to know about its limitation so that we can take advantage of its functionality.

## FAQs on C Array

### 1. Define Array in C.

An array is a fixed-size homogeneous collection of elements that are stored in a contiguous memory location.

### 2. How to declare an array in C?

We can declare array in C using the following syntax:

```
datatype array_name [size];
```

### 3. How do you initialize an array in C?

We can initialize an array using two methods:

- Using Initializer list
- Using Loops

#### Using Initializer List

We can use an initializer list to initialize the array with the declaration using the following syntax:

```
datatype array_name [size] = {value1, value2,...valueN};
```

## Using Loops

We can initialize an Array using Loops in C after the array declaration:

```
for (int i = 0; i < N; i++) {
    array_name[i] = valuei;
}
```

## 4. Why do we need Arrays?

We can use normal variables ( $v_1, v_2, v_3, \dots$ ) when we have a small number of objects, but if we want to store a large number of instances, it becomes difficult to manage them with normal variables. The idea of an array is to represent many instances in one variable.

## 5. How can we determine the size of the C array?

We can determine the size of the Array using sizeof Operator in C. We first get the size of the whole array and divide it by the size of each element type.

## 6. What is the difference between Arrays and Pointers?

The following table list the [differences between an array and a pointer](#):

Pointer	Array
A pointer is a derived data type that can store the address of other variables.	An array is a homogeneous collection of items of any type such as int, char, etc.
Pointers are allocated at run time.	Arrays are allocated at runtime.
The pointer is a single variable.	An array is a collection of variables of the same type.
Dynamic in Nature	Static in Nature.

**SALE!**  
**GeeksforGeeks Courses Upto 25% Off Enroll Now!**



Save 25% on Courses   DSA   Data Structures   Algorithms   Interview Preparation   Data Science   T

# Multidimensional Arrays in C

Difficulty Level : Easy • Last Updated : 21 Mar, 2023

[Read](#)   [Discuss](#)   [Courses](#)   [Practice](#)   [Video](#)

## Prerequisite: [Arrays in C](#)

A multi-dimensional array can be termed as an array of arrays that stores homogeneous data in tabular form. Data in multidimensional arrays is generally stored in row-major order in the memory.

The ***general form of declaring N-dimensional arrays*** is shown below.

## Syntax:

AD

```
data_type array_name[size1][size2]....[sizeN];
```

- **data\_type**: Type of data to be stored in the array.
- **array\_name**: Name of the array.
- **size1, size2,..., sizeN**: Size of each dimension.

## Examples:

```
Two dimensional array: int two_d[10][20];
```

```
Three dimensional array: int three_d[10][20][30];
```

## Size of Multidimensional Arrays:

The total number of elements that can be stored in a multidimensional array can be calculated by multiplying the size of all the dimensions.

### For example:

- The array **int x[10][20]** can store total  $(10 * 20) = 200$  elements.
- Similarly array **int x[5][10][20]** can store total  $(5 * 10 * 20) = 1000$  elements.

To get the size of the array in bytes, we multiply the size of a single element with the total number of elements in the array.

### For example:

- Size of array **int x[10][20]** =  $10 * 20 * 4 = 800$  bytes. (where int = 4 bytes)
- Similarly, size of **int x[5][10][20]** =  $5 * 10 * 20 * 4 = 4000$  bytes. (where int = 4 bytes)

The most commonly used forms of the multidimensional array are:

1. **Two Dimensional Array**
2. **Three Dimensional Array**

## Two-Dimensional Array in C

A **two-dimensional array** or **2D array** in C is the simplest form of the multidimensional array. We can visualize a two-dimensional array as an array of one-dimensional arrays arranged one over another forming a table with 'x' rows and 'y' columns where the row number ranges from 0 to  $(x-1)$  and the column number ranges from 0 to  $(y-1)$ .

	<b>Column 0</b>	<b>Column 1</b>	<b>Column 2</b>
<b>Row 0</b>	<b>x[0][0]</b>	<b>x[0][1]</b>	<b>x[0][2]</b>
<b>Row 1</b>	<b>x[1][0]</b>	<b>x[1][1]</b>	<b>x[1][2]</b>
<b>Row 2</b>	<b>x[2][0]</b>	<b>x[2][1]</b>	<b>x[2][2]</b>

*Graphical Representation of Two-Dimensional Array of Size 3 x 3*

## Declaration of Two-Dimensional Array in C

The basic form of declaring a 2D array with **x** rows and **y** columns in C is shown below.

### Syntax:

```
data_type array_name[x][y];
```

where,

- **data\_type**: Type of data to be stored in each element.
- **array\_name**: name of the array
- **x**: Number of rows.
- **y**: Number of columns.

We can declare a two-dimensional integer array say 'x' with 10 rows and 20 columns as:

### Example:

```
int x[10][20];
```

*Note: In this type of declaration, the array is allocated memory in the stack and the size of the array should be known at the compile time i.e. size of the array is fixed. We can also create an array dynamically in C by using methods mentioned [here](#).*

## Initialization of Two-Dimensional Arrays in C

The various ways in which a 2D array can be initialized are as follows:

1. **Using Initializer List**
2. **Using Loops**

### 1. Initialization of 2D array using Initializer List

We can initialize a 2D array in C by using an initializer list as shown in the example below.

#### First Method:

```
int x[3][4] = {0, 1 ,2 ,3 ,4 , 5 , 6 , 7 , 8 , 9 , 10 , 11}
```

The above array has 3 rows and 4 columns. The elements in the braces from left to right are stored in the table also from left to right. The elements will be filled in the array in order:

the first 4 elements from the left will be filled in the first row, the next 4 elements in the second row, and so on.

### **Second Method (better):**

```
int x[3][4] = {{0,1,2,3}, {4,5,6,7}, {8,9,10,11}};
```

This type of initialization makes use of nested braces. Each set of inner braces represents one row. In the above example, there is a total of three rows so there are three sets of inner braces. The advantage of this method is that it is easier to understand.

*Note: The number of elements in initializer list should always be less than or equal to the total number of elements in the array.*

## **2. Initialization of 2D array using Loops**

We can use any C loop to initialize each member of a 2D array one by one as shown in the below example.

### **Example:**

```
int x[3][4];

for(int i = 0; i < 3; i++){
    for(int j = 0; j < 4; j++){
        x[i][j] = i + j;
    }
}
```

This method is useful when the values of each element have some sequential relation.

## **Accessing Elements of Two-Dimensional Arrays in C**

Elements in 2D arrays are accessed using row indexes and column indexes. Each element in a 2D array can be referred to by:

### **Syntax:**

```
array_name[i][j]
```

where,

- **i:** The row index.

- **j:** The column index.

### Example:

```
int x[2][1];
```

The above example represents the element present in the third row and second column.

**Note:** In arrays, if the size of an array is N. Its index will be from 0 to N-1. Therefore, for row index 2 row number is  $2+1 = 3$ . To output all the elements of a Two-Dimensional array we can use nested for loops. We will require two 'for' loops. One to traverse the rows and another to traverse columns.

For printing the whole array, we access each element one by one using loops. The order of traversal can be row-major order or column-major order depending upon the requirement. The below example demonstrates the row-major traversal of a 2D array.

### Example:

---

## C

```
// C Program to print the elements of a
// Two-Dimensional array

#include <stdio.h>

int main(void)
{
    // an array with 3 rows and 2 columns.
    int x[3][2] = { { 0, 1 }, { 2, 3 }, { 4, 5 } };

    // output each array element's value
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 2; j++) {
            printf("Element at x[%i][%i]: ", i, j);
            printf("%d\n", x[i][j]);
        }
    }

    return (0);
}

// This code is contributed by sarajadhav12052009
```

### Output

```

Element at x[0][0]: 0
Element at x[0][1]: 1
Element at x[1][0]: 2
Element at x[1][1]: 3
Element at x[2][0]: 4
Element at x[2][1]: 5

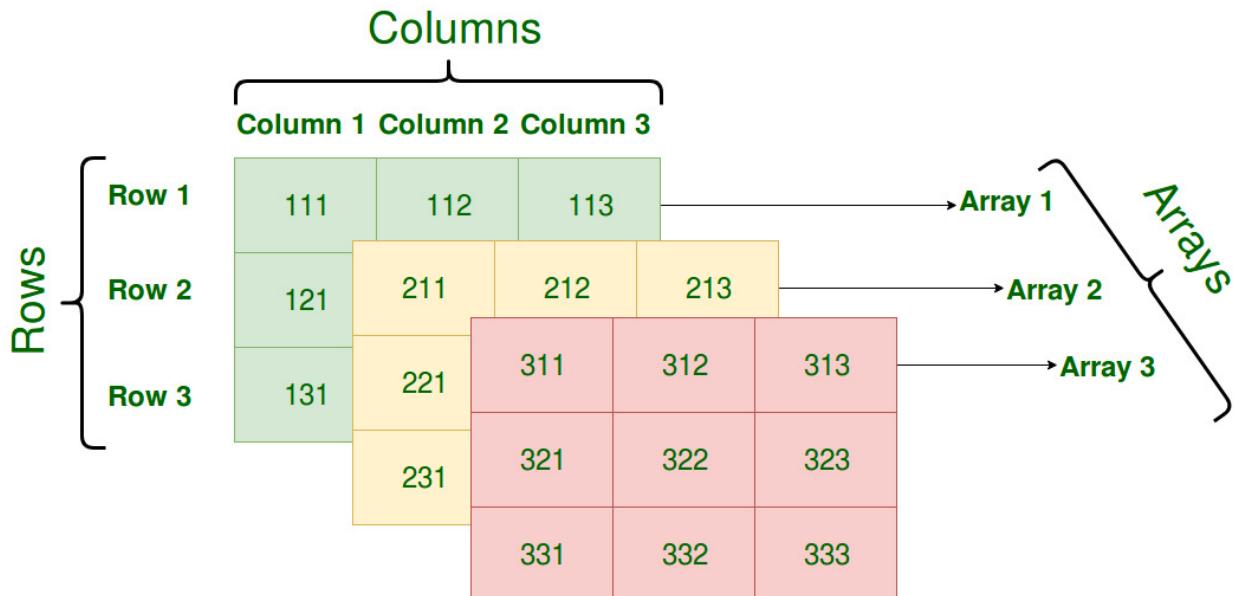
```

**Time Complexity:**  $O(N \times M)$ , where N(here 3) and M(here 2) are number of rows and columns respectively.

**Space Complexity:**  $O(1)$

## Three-Dimensional Array in C

A **Three Dimensional Array** or **3D** array in C is a collection of two-dimensional arrays. It can be visualized as multiple 2D arrays stacked on top of each other.



Graphical Representation of Three-Dimensional Array of Size  $3 \times 3 \times 3$

## Declaration of Three-Dimensional Array in C

We can declare a 3D array with **x** 2D arrays each having **y** rows and **z** columns using the syntax shown below.

### Syntax:

```
data_type array_name[x][y][z];
```

- **data\_type:** Type of data to be stored in each element.
- **array\_name:** name of the array

- **x:** Number of 2D arrays.
- **y:** Number of rows in each 2D array.
- **z:** Number of columns in each 2D array.

### **Example:**

```
int array[3][3][3];
```

## **Initialization of Three-Dimensional Array in C**

Initialization in a 3D array is the same as that of 2D arrays. The difference is as the number of dimensions increases so the number of nested braces will also increase.

A 3D array in C can be initialized by using:

1. **Initializer List**
2. **Loops**

### **Initialization of 3D Array using Initializer List**

#### **Method 1:**

```
int x[2][3][4] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
                  11, 12, 13, 14, 15, 16, 17, 18, 19,
                  20, 21, 22, 23};
```

#### **Method 2(Better):**

```
int x[2][3][4] =
{
    { {0,1,2,3}, {4,5,6,7}, {8,9,10,11} },
    { {12,13,14,15}, {16,17,18,19}, {20,21,22,23} }
};
```

### **Initialization of 3D Array using Loops**

It is also similar to that of 2D array with one more nested loop for accessing one more dimension.

```
int x[2][3][4];

for (int i=0; i<2; i++) {
    for (int j=0; j<3; j++) {
        for (int k=0; k<4; k++) {
            x[i][j][k] = (some_value);
```

```

    }
}
}
```

## Accessing elements in Three-Dimensional Array in C

Accessing elements in 3D Arrays is also similar to that of 3D Arrays. The difference is we have to use three loops instead of two loops for one additional dimension in 3D Arrays.

### Syntax:

```
array_name[x][y][z]
```

where,

- **x**: Index of 2D array.
- **y**: Index of that 2D array row.
- **z**: Index of that 2D array column.

## C

```
// C program to print elements of Three-Dimensional Array

#include <stdio.h>

int main(void)
{
    // initializing the 3-dimensional array
    int x[2][3][2] = { { { 0, 1 }, { 2, 3 }, { 4, 5 } },
                       { { 6, 7 }, { 8, 9 }, { 10, 11 } } };

    // output each element's value
    for (int i = 0; i < 2; ++i) {
        for (int j = 0; j < 3; ++j) {
            for (int k = 0; k < 2; ++k) {
                printf("Element at x[%i][%i][%i] = %d\n", i,
                       j, k, x[i][j][k]);
            }
        }
    }
    return (0);
}
```

## Output

```
Element at x[0][0][0] = 0
Element at x[0][0][1] = 1
```

```
Element at x[0][1][0] = 2
Element at x[0][1][1] = 3
Element at x[0][2][0] = 4
Element at x[0][2][1] = 5
Element at x[1][0][0] = 6
Element at x[1][0][1] = 7
Element at x[1][1][0] = 8
Element at x[1][1][1] = 9
Element at x[1][2][0] = 10
Element at x[1][2][1] = 11
```

In similar ways, we can create arrays with any number of dimensions. However, the complexity also increases as the number of dimensions increases. The most used multidimensional array is the Two-Dimensional Array.

Arrays are also closely related to pointers in C language. To know more about the Relationship of Arrays with Pointers in C, refer to [this](#) article.

460

## Related Articles

1. [Initialization of a multidimensional arrays in C/C++](#)

---
2. [Multidimensional Pointer Arithmetic in C/C++](#)

---
3. [How to print dimensions of multidimensional array in C++](#)

---
4. [Difference between multidimensional array in C++ and Java](#)

---
5. [Variable Length Arrays in C/C++](#)

---
6. [How Arrays are Passed to Functions in C/C++?](#)

---
7. [How to quickly swap two arrays of same size in C++?](#)

---
8. [Why is a\[i\] == i\[a\] in C/C++ arrays?](#)

---
9. [Associative arrays in C++](#)

---
10. [How to join two Arrays using STL in C++?](#)

SALE!  
GeeksforGeeks Courses Upto 25% Off Enroll Now!



Save 25% on Courses DSA Data Structures Algorithms Interview Preparation Data Science T

# How to print size of array parameter in C++?

Difficulty Level : Medium • Last Updated : 10 Mar, 2023

Read Discuss Courses Practice Video

## How to compute the size of an array CPP?

### C++

```
// A C++ program to show that it is wrong to
// compute size of an array parameter in a function
#include <iostream>
using namespace std;

void findSize(int arr[])
{
    cout << sizeof(arr) << endl;
}

int main()
{
    int a[10];
    cout << sizeof(a) << " ";
    findSize(a);
    return 0;
}
```

### Output

40 8

**Time Complexity:** O(1)

**Auxiliary Space:** O(n) where n is the size of the array.

The above output is for a machine where the size of an integer is 4 bytes and the size of a pointer is 8 bytes.

The **cout** statement inside main prints 40, and **cout** in **findSize** prints 8. The reason is, arrays are always passed pointers in functions, i.e., **findSize(int arr[])** and **findSize(int \*arr)** mean exactly same thing. Therefore the cout statement inside **findSize()** prints the size of a pointer. See [this](#) and [this](#) for details.

## How to find the size of an array in function?

We can pass a 'reference to the array'.

AD

---

## CPP

```
// A C++ program to show that we can use reference to
// find size of array
#include <iostream>
using namespace std;

void findSize(int (&arr)[10])
{
    cout << sizeof(arr) << endl;
}

int main()
{
    int a[10];
    cout << sizeof(a) << " ";
    findSize(a);
    return 0;
}
```

## Output

40 40

**Time Complexity:** O(1)

**Space Complexity:** O(n) where n is the size of array.

The above program doesn't look good as we have a hardcoded size of the array parameter.

**We can do it better using templates in C++.**

---

## CPP

```
// A C++ program to show that we use template and
// reference to find size of integer array parameter
#include <iostream>
using namespace std;

template <size_t n>
void findSize(int (&arr)[n])
{
    cout << sizeof(int) * n << endl;
}

int main()
{
    int a[10];
    cout << sizeof(a) << " ";
    findSize(a);
    return 0;
}
```

## Output

40 40

**Time Complexity:** O(1)

**Space Complexity:** O(n) where n is the size of array.

**We can make a generic function as well:**

---

## CPP

```
// A C++ program to show that we use template and
// reference to find size of any type array parameter
#include <iostream>
using namespace std;

template <typename T, size_t n>
void findSize(T (&arr)[n])
{
    cout << sizeof(T) * n << endl;
}
```

```

int main()
{
    int a[10];
    cout << sizeof(a) << " ";
    findSize(a);

    float f[20];
    cout << sizeof(f) << " ";
    findSize(f);
    return 0;
}

```

## Output

40 40  
80 80

**Time Complexity:** O(1)

**Space Complexity:** O(n)

**Now the next step is to print the size of a dynamically allocated array.**

It's your task man! I'm giving you a hint.

---

## CPP

```

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

int main()
{
    int *arr = (int*)malloc(sizeof(int) * 20);
    return 0;
}

```

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

**SALE!**  
**GeeksforGeeks Courses Upto 25% Off Enroll Now!**



Save 25% on Courses   DSA   Data Structures   Algorithms   Interview Preparation   Data Science   T

# std::string class in C++

Difficulty Level : Easy • Last Updated : 17 Feb, 2023

[Read](#)   [Discuss](#)   [Courses](#)   [Practice](#)   [Video](#)

C++ has in its definition a way to represent a **sequence of characters as an object of the class**. This class is called std:: string. The string class stores the characters as a sequence of bytes with the functionality of allowing **access to the single-byte character**.

## String vs Character Array

String	Char Array
A string is a <b>class that defines objects</b> that be represented as a stream of characters.	A character array is simply an <b>array of characters</b> that can be terminated by a null character.
In the case of strings, memory is <b>allocated dynamically</b> . More memory can be allocated at run time on demand. As no memory is preallocated, <b>no memory is wasted</b> .	The size of the character array has to be <b>allocated statically</b> , more memory cannot be allocated at run time if required. Unused allocated <b>memory is also wasted</b>
As strings are represented as objects, <b>no array decay</b> occurs.	There is a <b>threat of array decay</b> in the case of the character array.
<b>Strings are slower</b> when compared to implementation than character array.	Implementation of <b>character array is faster</b> than std:: string.
String class defines <b>a number of functionalities</b> that allow manifold	Character arrays <b>do not offer</b> many <b>inbuilt functions</b> to manipulate strings.

<b>String</b>	<b>Char Array</b>
operations on strings.	

## Operations on Strings

### 1) Input Functions

<b>Function</b>	<b>Definition</b>
<a href="#"><u>getline()</u></a>	This function is used to store a stream of characters as entered by the user in the object memory.
<a href="#"><u>push_back()</u></a>	This function is used to input a character at the end of the string.
<a href="#"><u>pop_back()</u></a>	Introduced from C++11(for strings), this function is used to delete the last character from the string.

### Example:

AD

## CPP

```
// C++ Program to demonstrate the working of
// getline(), push_back() and pop_back()
#include <iostream>
#include <string> // for string class
using namespace std;

// Driver Code
int main()
{
    // Declaring string
    string str;

    // Taking string input using getline()
    cout << "Enter a string: ";
    getline(cin, str);

    // Displaying string
    cout << "The string is: " << str;
}
```

```

getline(cin, str);

// Displaying string
cout << "The initial string is : ";
cout << str << endl;

// Inserting a character
str.push_back('s');

// Displaying string
cout << "The string after push_back operation is : ";
cout << str << endl;

// Deleting a character
str.pop_back();

// Displaying string
cout << "The string after pop_back operation is : ";
cout << str << endl;

return 0;
}

```

## Output

```

The initial string is :
The string after push_back operation is : s
The string after pop_back operation is :

```

## Time Complexity: O(1)

**Space Complexity: O(n)** where n is the size of the string

## 2) Capacity Functions

Function	Definition
capacity()	This function returns the capacity allocated to the string, which can be equal to or more than the size of the string. Additional space is allocated so that when the new characters are added to the string, the operations can be done efficiently.
<u>resize()</u>	This function changes the size of the string, the size can be increased or decreased.
length()	This function finds the length of the string.

Function	Definition
shrink_to_fit()	This function decreases the capacity of the string and makes it equal to the minimum capacity of the string. This operation is useful to save additional memory if we are sure that no further addition of characters has to be made.

**Example:****CPP**

```
// C++ Program to demonstrate the working of
// capacity(), resize() and shrink_to_fit()
#include <iostream>
#include <string> // for string class
using namespace std;

// Driver Code
int main()
{
    // Initializing string
    string str = "geeksforgeeks is for geeks";

    // Displaying string
    cout << "The initial string is : ";
    cout << str << endl;

    // Resizing string using resize()
    str.resize(13);

    // Displaying string
    cout << "The string after resize operation is : ";
    cout << str << endl;

    // Displaying capacity of string
    cout << "The capacity of string is : ";
    cout << str.capacity() << endl;

    // Displaying length of the string
    cout << "The length of the string is :" << str.length()
        << endl;

    // Decreasing the capacity of string
    // using shrink_to_fit()
    str.shrink_to_fit();

    // Displaying string
    cout << "The new capacity after shrinking is : ";
    cout << str.capacity() << endl;

    return 0;
}
```

## Output

```
The initial string is : geeksforgeeks is for geeks
The string after resize operation is : geeksforgeeks
The capacity of string is : 26
The length of the string is :13
The new capacity after shrinking is : 13
```

**Time Complexity:** O(1)

**Space Complexity:** O(n) where n is the size of the string

## 3) Iterator Functions

Function	Definition
begin()	This function returns an iterator to the beginning of the string.
end()	This function returns an iterator to the next to the end of the string.
rbegin()	This function returns a reverse iterator pointing at the end of the string.
rend()	This function returns a reverse iterator pointing to the previous of beginning of the string.
cbegin()	This function returns a constant iterator pointing to the beginning of the string, it cannot be used to modify the contents it points-to.
cend()	This function returns a constant iterator pointing to the next of end of the string, it cannot be used to modify the contents it points-to.
crbegin()	This function returns a constant reverse iterator pointing to the end of the string, it cannot be used to modify the contents it points-to.
crend()	This function returns a constant reverse iterator pointing to the previous of beginning of the string, it cannot be used to modify the contents it points-to.

### Algorithm:

1. Declare a string
2. Try to iterate the string using all types of iterators
3. Try modification of the element of the string.
4. Display all the iterations.

## Example:

---

### CPP

```
// C++ Program to demonstrate the working of
// begin(), end(), rbegin(), rend(), cbegin(), cend(), crbegin(), crend()
#include <iostream>
#include <string> // for string class
using namespace std;

// Driver Code
int main()
{
    // Initializing string
    string str = "geeksforgeeks";

    // Declaring iterator
    std::string::iterator it;

    // Declaring reverse iterator
    std::string::reverse_iterator it1;
    cout<<"Str:"<<str<<"\n";
    // Displaying string
    cout << "The string using forward iterators is : ";
    for (it = str.begin(); it != str.end(); it++){
        if(it == str.begin()) *it='G';
        cout << *it;
    }
    cout << endl;

    str = "geeksforgeeks";
    // Displaying reverse string
    cout << "The reverse string using reverse iterators is "
        ": ";
    for (it1 = str.rbegin(); it1 != str.rend(); it1++){
        if(it1 == str.rbegin()) *it1='S';
        cout << *it1;
    }
    cout << endl;

    str = "geeksforgeeks";
    //Displaying String
    cout<<"The string using constant forward iterator is :";
    for(auto it2 = str.cbegin(); it2!=str.cend(); it2++){
        //if(it2 == str.cbegin()) *it2='G';
        //here modification is NOT Possible
```

```

//error: assignment of read-only location
//As it is a pointer to the const content, but we can inc/dec-rement the iterator
cout<<*it2;
}

cout<<"\n";

str = "geeksforgeeks";
//Displaying String in reverse
cout<<"The reverse string using constant reverse iterator is :";
for(auto it3 = str.crbegin(); it3!=str.crend(); it3++){
    //if(it2 == str.begin()) *it2='S';
    //here modification is NOT Possible
    //error: assignment of read-only location
    //As it is a pointer to the const content, but we can inc/dec-rement the iterator
    cout<<*it3;
}
cout<<"\n";

return 0;
}

//Code modified by Balakrishnan R (rbkraj000)

```

## Output

Str:geeksforgeeks  
The string using forward iterators is : Geeksforgeeks  
The reverse string using reverse iterators is : Skeegrofskeeg  
The string using constant forward iterator is :geeksforgeeks  
The reverse string using constant reverse iterator is :skeegrofskeeg

**Time Complexity: O(1)**

**Space Complexity: O(n)** where n is the size of the string

## 4) Manipulating Functions:

Function	Definition
copy("char array", len, pos)	This function copies the substring in the target character array mentioned in its arguments. It takes 3 arguments, target char array, length to be copied, and starting position in the string to start copying.
swap()	This function swaps one string with another

## Example:

## CPP

```
// C++ Program to demonstrate the working of
// copy() and swap()
#include <iostream>
#include <string> // for string class
using namespace std;

// Driver Code
int main()
{
    // Initializing 1st string
    string str1 = "geeksforgeeks is for geeks";

    // Declaring 2nd string
    string str2 = "geeksforgeeks rocks";

    // Declaring character array
    char ch[80];

    // using copy() to copy elements into char array
    // copies "geeksforgeeks"
    str1.copy(ch, 13, 0);

    // Displaying char array
    cout << "The new copied character array is : ";
    cout << ch << endl;

    // Displaying strings before swapping
    cout << "The 1st string before swapping is : ";
    cout << str1 << endl;
    cout << "The 2nd string before swapping is : ";
    cout << str2 << endl;

    // using swap() to swap string content
    str1.swap(str2);

    // Displaying strings after swapping
    cout << "The 1st string after swapping is : ";
    cout << str1 << endl;
    cout << "The 2nd string after swapping is : ";
    cout << str2 << endl;

    return 0;
}
```

## Output

The new copied character array is : geeksforgeeks  
The 1st string before swapping is : geeksforgeeks is for geeks  
The 2nd string before swapping is : geeksforgeeks rocks

The 1st string after swapping is : geeksforgeeks rocks

The 2nd string after swapping is : geeksforgeeks is for geeks

### Must Read: [C++ String Class and its Applications](#)

C++ Programming Language Tutorial | Strings in C++ | GeeksforGeeks



This article is contributed by [Manjeet Singh](#). If you like GeeksforGeeks and would like to contribute, you can also write an article using [write.geeksforgeeks.org](#) or mail your article to review-team@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks. Please write comments if you find anything incorrect, or if you want to share more information about the topic discussed above.

480

## Related Articles

1. Behavior of virtual function in the derived class from the base class and abstract class
2. How to convert a class to another class type in C++?
3. Base Class Pointer Pointing to Derived Class Object in C++
4. Difference between Base class and Derived class in C++
5. Can a C++ class have an object of self type?
6. Hiding of all Overloaded Methods with Same Name in Base Class in C++
7. Why is the Size of an Empty Class Not Zero in C++?
8. Simulating final Class in C++
9. What happens when more restrictive access is given to a derived class method in C++?

SALE!  
GeeksforGeeks Courses Upto 25% Off Enroll Now!



Save 25% on Courses DSA Data Structures Algorithms Interview Preparation Data Science T

# Raw String Literal in C++

Difficulty Level : Easy • Last Updated : 27 Jan, 2023

[Read](#) [Discuss](#) [Courses](#) [Practice](#) [Video](#)

A Literal is a constant variable whose value does not change during the lifetime of the program. Whereas, a raw string literal is a string in which the escape characters like '\n, \t, or \"' of C++ are not processed. Hence, a raw string literal that starts with **R"( and ends in )"**.

## The syntax for Raw string Literal:

```
R "delimiter( raw_characters )delimiter" // delimiter is the end of logical entity
```

Here, delimiter is optional and it can be a character except the backslash{ / }, whitespaces{ }, and parentheses { () }.

These raw string literals allow a series of characters by writing precisely its contents like raw character sequence.

AD

## Example:

### Ordinary String Literal

```
"\\\\\\n"
```

## Raw String Literal

```
\!-- Delimiter
R"(\n)"
/\!-- Delimiter
```

### Difference between an Ordinary String Literal and a Raw String Literal:

Ordinary String Literal	Raw String Literal
It does not need anything to be defined.	It needs a defined line{ parentheses ()} to start with the prefix <b>R</b> .
It does not allow/include nested characters.	It allows/includes nested character implementation.
It does not ignore any special meaning of character and implements their special characteristic.	It ignores all the special characters like <b>\n</b> and <b>\t</b> and treats them like normal text.

### Example of Raw String Literal:

#### CPP

```
// C++ program to demonstrate working of raw string literal
#include <iostream>
using namespace std;

// Driver Code
int main()
{
    // A Normal string
    string string1 = "Geeks.\nFor.\nGeeks.\n";

    // A Raw string
    string string2 = R"(Geeks.\nFor.\nGeeks.\n)";

    cout << string1 << endl;
    cout << string2 << endl;

    return 0;
}
```

## Output

Geeks.  
For.  
Geeks.

Geeks.\nFor.\nGeeks.\n

### Time Complexity: O(n)

### Space complexity: O(n)

This article is contributed by **MAZHAR IMAM KHAN**. If you like GeeksforGeeks and would like to contribute, you can also write an article using [write.geeksforgeeks.org](https://www.geeksforgeeks.org/write.geeksforgeeks.org) or mail your article to review-team@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks. Please write comments if you find anything incorrect, or if you want to share more information about the topic discussed above.

144

## Related Articles

1. [How to Print String Literal and QString With QDebug in C++?](#)

---
2. [ASCII NULL, ASCII 0 \('0'\) and Numeric literal 0](#)

---
3. [Integer literal in C/C++ \(Prefixes and Suffixes\)](#)

---
4. [Multi-Character Literal in C/C++](#)

---
5. [std::string::length, std::string::capacity, std::string::size in C++ STL](#)

---
6. [std::string::replace , std::string::replace\\_if in C++](#)

---
7. [std::string::replace\\_copy\(\), std::string::replace\\_copy\\_if in C++](#)

---
8. [std::string::append vs std::string::push\\_back\(\) vs Operator += in C++](#)

---
9. [std::string::remove\\_copy\(\), std::string::remove\\_copy\\_if\(\) in C++](#)

---
10. [Check if a string can be formed from another string using given constraints](#)



SALE!

GeeksforGeeks Courses Upto 25% Off Enroll Now!

[Save 25% on Courses](#) DSA Data Structures Algorithms Array Strings Linked List Stack Q

# Array of Strings in C++ – 5 Different Ways to Create

Difficulty Level : Easy • Last Updated : 11 Mar, 2023

[Read](#)[Discuss\(20+\)](#)[Courses](#)[Practice](#)[Video](#)

In C++, a string is usually just an array of (or a reference/points to) characters that ends with the NULL character '\0'. A string is a 1-dimensional array of characters and an array of strings is a 2-dimensional array of characters where each row contains some string.

**Below are the 5 different ways to create an Array of Strings in C++:**

1. Using **Pointers**
2. Using **2-D Array**
3. Using the **String Class**
4. Using the **Vector Class**
5. Using the **Array Class**

## 1. Using Pointers

Pointers are the symbolic representation of an address. In simple words, a pointer is something that stores the address of a variable in it. In this method, an array of string literals is created by an array of pointers in which each pointer points to a particular string.

**Example:**

AD

## C++

```
// C++ program to demonstrate
// array of strings using
// pointers character array
#include <iostream>

// Driver code
int main()
{
    // Initialize array of pointer
    const char* colour[4]
        = { "Blue", "Red", "Orange", "Yellow" };

    // Printing Strings stored in 2D array
    for (int i = 0; i < 4; i++)
        std::cout << colour[i] << "\n";

    return 0;
}
```

## Output

Blue  
Red  
Orange  
Yellow

### Explanation:

- The number of strings is fixed, but needn't be. The 4 may be omitted, and the compiler will compute the correct size.
- These strings are constants and their contents cannot be changed. Because string literals (literally, the quoted strings) exist in a read-only area of memory, we must specify "const" here to prevent unwanted accesses that may crash the program.

## 2. Using a 2D array

A [2-D array](#) is the simplest form of a multidimensional array in which it stores the data in a tabular form. This method is useful when the length of all strings is known and a particular memory footprint is desired. Space for strings will be allocated in a single block

### Example:

---

#### C++

```
// C++ program to demonstrate
// array of strings using
// 2D character array
#include <iostream>

// Driver code
int main()
{
    // Initialize 2D array
    char colour[4][10]
        = { "Blue", "Red", "Orange", "Yellow" };

    // Printing Strings stored in 2D array
    for (int i = 0; i < 4; i++)
        std::cout << colour[i] << "\n";

    return 0;
}
```

#### Output

```
Blue
Red
Orange
Yellow
```

#### Explanation:

- Both the number of strings and the size of the strings are fixed. The 4, again, may be left out, and the appropriate size will be computed by the compiler. The second dimension, however, must be given (in this case, 10), so that the compiler can choose an appropriate memory layout.
- Each string can be modified but will take up the full space given by the second dimension. Each will be laid out next to each other in memory, and can't change size.
- Sometimes, control over the memory footprint is desirable, and this will allocate a region of memory with a fixed, regular layout.

## 3. Using the String class

The STL [string](#) or [string class](#) may be used to create an array of mutable strings. In this method, the size of the string is not fixed, and the strings can be changed which somehow makes it dynamic in nature nevertheless **std::string** can be used to create a string array using in-built functions.

### Example:

---

#### C++

```
// C++ program to demonstrate
// array of strings using
// string class
#include <iostream>
#include <string>

// Driver code
int main()
{
    // Initialize String Array
    std::string colour[4]
        = { "Blue", "Red", "Orange", "Yellow" };

    // Print Strings
    for (int i = 0; i < 4; i++)
        std::cout << colour[i] << "\n";
}
```

#### Output

```
Blue
Red
Orange
Yellow
```

#### Explanation:

The array is of fixed size, but needn't be. Again, the 4 here may be omitted, and the compiler will determine the appropriate size of the array. The strings are also mutable, allowing them to be changed.

## 4. Using the vector class

A [vector](#) is a dynamic array that doubles its size whenever a new character is added that exceeds its limit. The STL container vector can be used to dynamically allocate an array that can vary in size.

This is only usable in C++, as C does not have classes. Note that the initializer-list syntax here requires a compiler that supports the 2011 C++ standard, and though it is quite likely your compiler does, it is something to be aware of.

### Example:

---

## C++

```
// C++ program to demonstrate
// array of strings using
// vector class
#include <iostream>
#include <string>
#include <vector>

// Driver code
int main()
{
    // Declaring Vector of String type
    // Values can be added here using
    // initializer-list
    // syntax
    std::vector<std::string> colour{"Blue", "Red",
                                    "Orange"};

    // Strings can be added at any time
    // with push_back
    colour.push_back("Yellow");

    // Print Strings stored in Vector
    for (int i = 0; i < colour.size(); i++)
        std::cout << colour[i] << "\n";
}
```

## Output

Blue  
Red  
Orange  
Yellow

### Explanation:

- Vectors are dynamic arrays and allow you to add and remove items at any time.
- Any type or class may be used in vectors, but a given vector can only hold one type.

## 5. Using the Array Class

An array is a homogeneous mixture of data that is stored continuously in the memory space.

The STL [container array](#) can be used to allocate a fixed-size array. It may be used very similarly to a vector, but the size is always fixed.

### Example:

---

## C++

```
// C++ program to demonstrate
// array of string using STL array
#include <array>
#include <iostream>
#include <string>

// Driver code
int main()
{
    // Initialize array
    std::array<std::string, 4> colour{"Blue", "Red",
                                      "Orange", "Yellow"};

    // Printing Strings stored in array
    for (int i = 0; i < 4; i++)
        std::cout << colour[i] << "\n";

    return 0;
}
```

## Output

```
Blue
Red
Orange
Yellow
```

These are by no means the only ways to make a collection of strings. C++ offers several [container](#) classes, each of which has various tradeoffs and features, and all of them exist to fill requirements that you will have in your projects. Explore and have fun!

**Conclusion:** Out of all the methods, Vector seems to be the best way for creating an array of Strings in C++.

This article is contributed by **Kartik Ahuja**. If you like GeeksforGeeks and would like to contribute, you can also write an article using [write.geeksforgeeks.org](https://write.geeksforgeeks.org). See your article appearing on the GeeksforGeeks main page and help other Geeks. Please write comments if you find anything incorrect, or if you want to share more information about the topic discussed above.s.

**SALE!**  
**GeeksforGeeks Courses Upto 25% Off Enroll Now!**



Save 25% on Courses   DSA   Data Structures   Algorithms   Array   Strings   Linked List   Stack   Q

# Tokenizing a string in C++

Difficulty Level : Medium • Last Updated : 02 Jan, 2023

[Read](#)   [Discuss](#)   [Courses](#)   [Practice](#)   [Video](#)

Tokenizing a string denotes splitting a string with respect to some delimiter(s). There are many ways to tokenize a string. In this article four of them are explained:

## Using stringstream

A **stringstream** associates a string object with a stream allowing you to read from the string as if it were a stream.

Below is the C++ implementation :

## C++

```
// Tokenizing a string using stringstream
#include <bits/stdc++.h>

using namespace std;

int main()
{
    string line = "GeeksForGeeks is a must try";

    // Vector of string to save tokens
    vector <string> tokens;

    // stringstream class check1
    stringstream check1(line);

    string intermediate;

    // Tokenizing w.r.t. space ' '
    while(getline(check1, intermediate, ' '))
        cout << intermediate << endl;
}
```

```

{
    tokens.push_back(intermediate);
}

// Printing the token vector
for(int i = 0; i < tokens.size(); i++)
    cout << tokens[i] << '\n';
}

```

## Output

AD

GeeksForGeeks  
is  
a  
must  
try

**Time Complexity:** O(n) where n is the length of string.

**Auxiliary Space:** O(n-d) where n is the length of string and d is the number of delimiters.

## Using strtok()

```

// Splits str[] according to given delimiters.
// and returns next token. It needs to be called
// in a loop to get all tokens. It returns NULL
// when there are no more tokens.
char * strtok(char str[], const char *delims);

```

Below is the C++ implementation :

## C++

```

// C/C++ program for splitting a string
// using strtok()
#include <stdio.h>
#include <string.h>

```

```

int main()
{
    char str[] = "Geeks-for-Geeks";

    // Returns first token
    char *token = strtok(str, "-");

    // Keep printing tokens while one of the
    // delimiters present in str[].
    while (token != NULL)
    {
        printf("%s\n", token);
        token = strtok(NULL, "-");
    }

    return 0;
}

```

## Output

Geeks  
for  
Geeks

**Time Complexity:** O(n) where n is the length of string.

**Auxiliary Space:** O(1).

**Another Example of strtok() :**

---

## C

```

// C code to demonstrate working of
// strtok
#include <string.h>
#include <stdio.h>

// Driver function
int main()
{
    // Declaration of string
    char gfg[100] = " Geeks - for - geeks - Contribute";

    // Declaration of delimiter
    const char s[4] = "-";
    char* tok;

    // Use of strtok
    // get first token
    tok = strtok(gfg, s);

    // Checks for delimiter

```

```

while (tok != 0) {
    printf(" %s\n", tok);

    // Use of strtok
    // go through other tokens
    tok = strtok(0, s);
}

return (0);
}

```

## Output

Geeks  
for  
geeks  
Contribute

**Time Complexity:** O(n ) where n is the length of string.

**Auxiliary Space:** O(1).

## Using strtok\_r()

Just like strtok() function in C, **strtok\_r()** does the same task of parsing a string into a sequence of tokens. strtok\_r() is a reentrant version of strtok().

There are two ways we can call strtok\_r()

```

// The third argument saveptr is a pointer to a char *
// variable that is used internally by strtok_r() in
// order to maintain context between successive calls
// that parse the same string.
char *strtok_r(char *str, const char *delim, char **saveptr);

```

Below is a simple C++ program to show the use of strtok\_r() :

## C++

```

// C/C++ program to demonstrate working of strtok_r()
// by splitting string based on space character.
#include<stdio.h>
#include<string.h>

int main()
{
    char str[] = "Geeks for Geeks";
    char *token;

```

```

char *rest = str;

while ((token = strtok_r(rest, " ", &rest)))
    printf("%s\n", token);

return(0);
}

```

## Output

Geeks  
for  
Geeks

**Time Complexity:** O(n) where n is the length of string.

**Auxiliary Space:** O(1).

## Using std::sregex\_token\_iterator

In this method the tokenization is done on the basis of regex matches. Better for use cases when multiple delimiters are needed.

Below is a simple C++ program to show the use of std::sregex\_token\_iterator:

## C++

```

// CPP program for above approach
#include <iostream>
#include <regex>
#include <string>
#include <vector>

/**
 * @brief Tokenize the given vector
 * according to the regex
 * and remove the empty tokens.
 *
 * @param str
 * @param re
 * @return std::vector<std::string>
 */
std::vector<std::string> tokenize(
    const std::string str,
    const std::regex re)
{
    std::sregex_token_iterator it{ str.begin(),
                                str.end(), re, -1 };
    std::vector<std::string> tokenized{ it, {} };

    // Additional check to remove empty strings

```

```

    tokenized.erase(
        std::remove_if(tokenized.begin(),
                      tokenized.end(),
                      [](std::string const& s) {
                          return s.size() == 0;
                      }),
        tokenized.end()));

    return tokenized;
}

// Driver Code
int main()
{
    const std::string str = "Break string
                           a,spaces,and,commas";
    const std::regex re(R"([\s|,]+)");

    // Function Call
    const std::vector<std::string> tokenized =
        tokenize(str, re);

    for (std::string token : tokenized)
        std::cout << token << std::endl;
    return 0;
}

```

## Output

Break  
 string  
 a  
 spaces  
 and  
 commas

**Time Complexity:**  $O(n * d)$  where n is the length of string and d is the number of delimiters.

**Auxiliary Space:**  $O(n)$

75

## Related Articles

1. Count characters of a string which when removed individually makes the string equal to another string

---

2. Generate string by incrementing character of given string by number present at corresponding index of second string

SALE!  
GeeksforGeeks Courses Upto 25% Off Enroll Now!



## strrchr() in C++

Save 25% on Courses DSA Data Structures Algorithms Interview Preparation Data Science T

Read Discuss Courses Practice Video

C++ `strrchr()` function finds the location of the **last occurrence** of the **specified** character in the given string and returns the pointer to it. It returns the NULL pointer if the character is not found.

It is a standard library function of C which is inherited by C++ so it only works on C-style strings (i.e. array of characters). It is defined inside `<cstring>` and `<string.h>` header files.

### Syntax:

```
char *strrchr(const char *str, int chr);
```

### Parameter:

AD

- **str:** specifies the pointer to the null-terminated string in which the search is to be performed.
- **chr:** specifies the character to be searched.

### Return Value:

- The function returns a pointer to the last location of **chr** in the string if the **chr** is found.
- If **chr** is not found, it returns a **NULL point**

**Recommended:** Please try your approach on [\(IDE\)](#) first, before moving on to the solution.

### Example:

#### C++

```
// C++ program to demonstrate working strchr()
#include <cstring>
#include <iostream>
using namespace std;

int main()
{
    char str[] = "This is a string";
    char* ch = strrchr(str, 'i');
    cout << "Index of last occurrence of i: "
        << ch - str + 1;
    return 0;
}
```

#### Output

9

**Time Complexity:**  $O(n)$ ,

**Space Complexity:**  $O(1)$ ,

where **n** is the length of the string.

### Practical Application of strrchr() function in C++

Since it returns the entire string after the last occurrence of a particular character, it can be used to **extract the suffix of a string**. For e.g to know the entire leading zeroes in a denomination when we know the first number.

### Example:

#### C++

```
// C++ code to demonstrate the application of
// strrchr()
#include <cstring>
#include <iostream>
using namespace std;
```

```
int main()
{
    // initializing the denomination
    char denom[] = "Rs 10000000";

    // Printing original string
    cout << "The original string is : " << denom;

    // initializing the initial number
    char first = '1';
    char* entire;

    // Use of strrchr()
    // returns entire number
    entire = strrchr(denom, first);

    cout << "\nThe denomination value is : " << entire;

    return 0;
}
```

## Output

```
The original string is : Rs 10000000
The denomination value is : 10000000
```

**Time Complexity:**  $O(N)$ , as time complexity for function `strrchr()` is  $O(N)$  where  $N$  is the length of given String .

**Auxiliary Space:**  $O(1)$ , since we are not using any extra space.

This article is contributed by **Ayush Saxena** and **Vaishnavi Tripathi**. If you like GeeksforGeeks and would like to contribute, you can also write an article using [write.geeksforgeeks.org](https://write.geeksforgeeks.org). See your article appearing on the GeeksforGeeks main page and help other Geeks. Please write comments if you find anything incorrect, or if you want to share more information about the topic discussed above.

36

## Related Articles

1. [C++ Error - Does not name a type](#)
2. [Execution Policy of STL Algorithms in Modern C++](#)
3. [C++ Program To Print Pyramid Patterns](#)

**SALE!**  
**GeeksforGeeks Courses Upto 25% Off Enroll Now!**



Save 25% on Courses   DSA   Data Structures   Algorithms   Array   Strings   Linked List   Stack   Q

# stringstream in C++ and its Applications

Difficulty Level : Medium • Last Updated : 28 Mar, 2023

[Read](#)   [Discuss](#)   [Courses](#)   [Practice](#)   [Video](#)

A stringstream associates a string object with a stream allowing you to read from the string as if it were a stream (like cin). To use stringstream, we need to include **sstream** header file. The stringstream class is extremely useful in parsing input.

## Basic methods are:

1. **clear()**- To clear the stream.
2. **str()**- To get and set string object whose content is present in the stream.
3. **operator <<**- Add a string to the stringstream object.
4. **operator >>**- Read something from the stringstream object.

## Examples:

### 1. Count the number of words in a string

AD

## Examples:

**Input:** Asipu Pawan Kumar

**Output:** 3

**Input:** Geeks For Geeks Ide

**Output:** 4

Below is the C++ program to implement the above approach--

## C++

```
// C++ program to count words in
// a string using stringstream.
#include <iostream>
#include <sstream>
#include<string>
using namespace std;

int countWords(string str)
{
    // Breaking input into word
    // using string stream

    // Used for breaking words
    stringstream s(str);

    // To store individual words
    string word;

    int count = 0;
    while (s >> word)
        count++;
    return count;
}

// Driver code
int main()
{
    string s = "geeks for geeks geeks "
               "contribution placements";
    cout << " Number of words are: " << countWords(s);
    return 0;
}
```

## Output

Number of words are: 6

**Time complexity:**  $O(n * \log(n))$ .

**Auxiliary space:**  $O(n)$ .

## 2. Print frequencies of individual words in a string

## Examples:

**Input:** Geeks For Geeks Quiz Geeks Quiz Practice Practice

**Output:** For -> 1

Geeks -> 3

Practice -> 2

Quiz -> 2

**Input:** Word String Frequency String

**Output:** Frequency -> 1

String -> 2

Word -> 1

Below is the C++ program to implement the above approach-

## C++

```
// C++ program to demonstrate use
// of stringstream to count
// frequencies of words.
#include <bits/stdc++.h>
#include <iostream>
#include <sstream>
#include<string>
using namespace std;

void printFrequency(string st)
{
    // Each word it mapped to
    // it's frequency
    map<string, int> FW;

    // Used for breaking words
    stringstream ss(st);

    // To store individual words
    string Word;

    while (ss >> Word)
        FW[Word]++;

    map<string, int>::iterator m;
    for (m = FW.begin(); m != FW.end(); m++)
        cout << m->first << "-> "
            << m->second << "\n";
}

// Driver code
```

```

int main()
{
    string s = "Geeks For Geeks Ide";
    printFrequency(s);
    return 0;
}

```

## Output

For-> 1  
 Geeks-> 2  
 Ide-> 1

**Time complexity:**  $O(n * \log(n))$ .

**Auxiliary space:**  $O(n)$ .

### 3. Convert Integer to string

Since, the insertion and extraction operators of string stream work with different data types. So that's why it works well with integers.

We will insert an integer into the string stream and after extracting that into a string, that integer value will become a string.

#### Code-

---

## C++

```

// C++ program to demonstrate the
// use of a stringstream to
// convert int to string
#include <iostream>
#include <sstream>
using namespace std;

// Driver code
int main()
{
    int val=123;

    // object from the class stringstream
    stringstream geek;

    // inserting integer val in geek stream
    geek << val;

    // The object has the value 123
    // and stream it to the string x
    string x;
    geek >> x;
}

```

```
// Now the string x holds the
// value 123
cout<<x+ "4" << endl;
return 0;
}
```

**Output-**

1234

**Time complexity:**  $O(n)$ , n is the length of the integer

**Auxiliary space:**  $O(n)$

[Removing spaces from a string using Stringstream](#)

[Converting Strings to Numbers in C/C++](#)

This article is contributed by **ASIPU PAWAN KUMAR**. If you like GeeksforGeeks and would like to contribute, you can also write an article using [write.geeksforgeeks.org](#) or mail your article to review-team@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

346

## Related Articles

1. [StringStream in C++ for Decimal to Hexadecimal and back](#)

---

2. [Removing spaces from a string using Stringstream](#)

---

3. [Finding number of days between two dates using stringstream](#)

---

4. [Find words which are greater than given length k using stringstream](#)

---

5. [C++ String Class and its Applications](#)

---

6. [strchr\(\) function in C++ and its applications](#)

---

7. [C++ string class and its applications](#)

---

8. [MakeFile in C++ and its applications](#)

---

9. [Different Types of Queues and its Applications](#)

**SALE!**  
**GeeksforGeeks Courses Upto 25% Off Enroll Now!**



Save 25% on Courses   DSA   Data Structures   Algorithms   Interview Preparation   Data Science   T

# Functions in C++

Difficulty Level : Easy • Last Updated : 16 Mar, 2023

[Read](#)   [Discuss](#)   [Courses](#)   [Practice](#)   [Video](#)

A function is a set of statements that take inputs, do some specific computation, and produce output. The idea is to put some commonly or repeatedly done tasks together and make a **function** so that instead of writing the same code again and again for different inputs, we can call the function.

In simple terms, a function is a block of code that only runs when it is called.

## Syntax:



*Syntax of Function*

## Example:

### C++

```

// C++ Program to demonstrate working of a function
#include <iostream>
using namespace std;

// Following function that takes two parameters 'x' and 'y'
// as input and returns max of two input numbers
int max(int x, int y)
{

```

```

if (x > y)
    return x;
else
    return y;
}

// main function that doesn't receive any parameter and
// returns integer
int main()
{
    int a = 10, b = 20;

    // Calling above function to find max of 'a' and 'b'
    int m = max(a, b);

    cout << "m is " << m;
    return 0;
}

```

## Output

AD

m is 20

**Time complexity:** O(1)

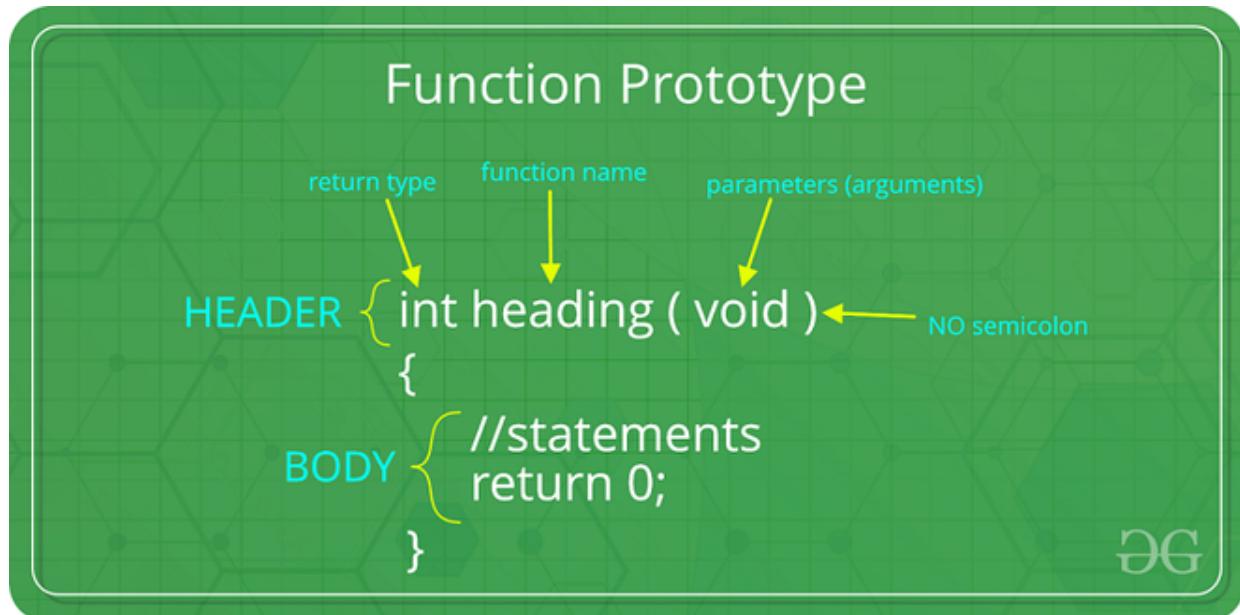
**Space complexity:** O(1)

## Why Do We Need Functions?

- Functions help us in **reducing code redundancy**. If functionality is performed at multiple places in software, then rather than writing the same code, again and again, we create a function and call it everywhere. This also helps in maintenance as we have to change at one place if we make future changes to the functionality.
- Functions make code **modular**. Consider a big file having many lines of code. It becomes really simple to read and use the code if the code is divided into functions.
- Functions provide **abstraction**. For example, we can use library functions without worrying about their internal work.

## Function Declaration

A function declaration tells the compiler about the number of parameters function takes data-types of parameters, and returns the type of function. Putting parameter names in the function declaration is optional in the function declaration, but it is necessary to put them in the definition. Below are an example of function declarations. (parameter names are not there in the below declarations)



Function Declaration

### Example:

## C++

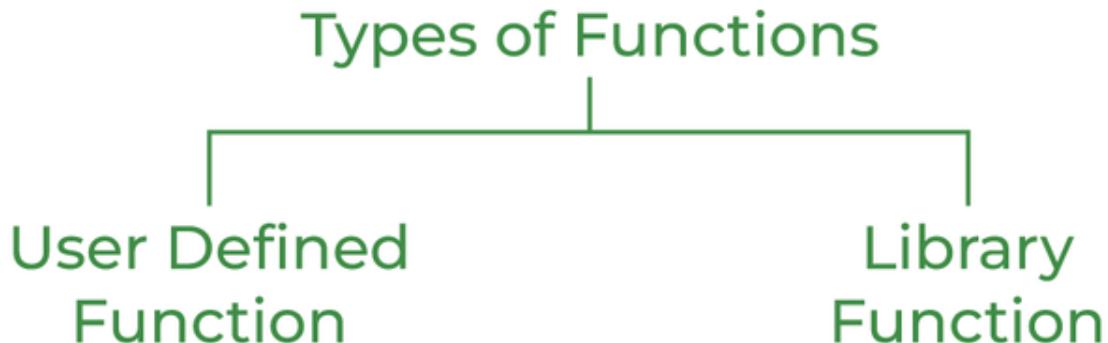
```
// C++ Program to show function that takes
// two integers as parameters and returns
// an integer
int max(int, int);

// A function that takes an int
// pointer and an int variable
// as parameters and returns
// a pointer of type int
int* swap(int*, int);

// A function that takes
// a char as parameter and
// returns a reference variable
char* call(char b);

// A function that takes a
// char and an int as parameters
// and returns an integer
int fun(char, int);
```

## Types of Functions



*Types of Function in C++*

### User Defined Function

User Defined functions are user/customer-defined blocks of code specially customized to reduce the complexity of big programs. They are also commonly known as "**tailor-made functions**" which are built only to satisfy the condition in which the user is facing issues meanwhile reducing the complexity of the whole program.

### Library Function

Library functions are also called "**builtin Functions**". These functions are a part of a compiler package that is already defined and consists of a special function with special and different meanings. Builtin Function gives us an edge as we can directly use them without defining them whereas in the user-defined function we have to declare and define a function before using them.

**For Example:** sqrt(), setw(), strcat(), etc.

## Parameter Passing to Functions

The parameters passed to function are called **actual parameters**. For example, in the program below, 5 and 10 are actual parameters.

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

```

class Multiplication {
    int multiply(int x, int y) { return x * y; }
public
    static void main()
    {
        Multiplication M = new Multiplication();
        int gfg = 5, gfg2 = 10;
        int gfg3 = multiply(gfg, gfg2);
        cout << "Result is " << gfg3;
    }
}

```

*Formal Parameter and Actual Parameter*

### There are two most popular ways to pass parameters:

1. **Pass by Value:** In this parameter passing method, values of actual parameters are copied to the 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 the actual parameters of the caller.
2. **Pass by Reference:** Both actual and formal parameters refer to the same locations, so any changes made inside the function are actually reflected in the actual parameters of the caller.

## Function Definition

**Pass by value** is used where the value of x is not modified using the function fun().

### C++

```

// C++ Program to demonstrate function definition
#include <iostream>
using namespace std;

void fun(int x)
{
    // definition of
    // function
    x = 30;
}

int main()
{
    int x = 20;
    fun(x);
}

```

```

cout << "x = " << x;
return 0;
}

```

## Output

x = 20

**Time complexity:** O(1)

**Space complexity:** O(1)

## Functions Using Pointers

The function fun() expects a pointer ptr to an integer (or an address of an integer). It modifies the value at the address ptr. The dereference operator \* is used to access the value at an address. In the statement '\*ptr = 30', the value at address ptr is changed to 30. The address operator & is used to get the address of a variable of any data type. In the function call statement 'fun(&x)', the address of x is passed so that x can be modified using its address.

---

## C++

```

// C++ Program to demonstrate working of
// function using pointers
#include <iostream>
using namespace std;

void fun(int* ptr) { *ptr = 30; }

int main()
{
    int x = 20;
    fun(&x);
    cout << "x = " << x;

    return 0;
}

```

## Output

x = 30

**Time complexity:** O(1)

**Space complexity:** O(1)

## Difference between call by value and call by reference in C++

Call by value	Call by reference
A copy of value is passed to the function	An address of value is passed to the function
Changes made inside the function is not reflected on other functions	Changes made inside the function is reflected outside the function also
Actual and formal arguments will be created in different memory location	Actual and formal arguments will be created in same memory location.

## Points to Remember About Functions in C++

1. Most C++ program has a function called `main()` that is called by the operating system when a user runs the program.
2. Every function has a return type. If a function doesn't return any value, then `void` is used as a return type. Moreover, if the return type of the function is `void`, we still can use the `return` statement in the body of the function definition by not specifying any constant, variable, etc. with it, by only mentioning the '`return;`' statement which would symbolize the termination of the function as shown below:

### C++

```
void function name(int a)
{
    ..... // Function Body
    return; // Function execution would get terminated
}
```

3. To declare a function that can only be called without any parameter, we should use "**void fun(void)**". As a side note, in C++, an empty list means a function can only be called without any parameter. In C++, both `void fun()` and `void fun(void)` are same.

## Main Function

The main function is a special function. Every C++ program must contain a function named `main`. It serves as the entry point for the program. The computer will start running the code

from the beginning of the main function.

## Types of Main Functions

### 1. Without parameters:

#### CPP

```
// Without Parameters
int main() { ... return 0; }
```

### 2. With parameters:

#### CPP

```
// With Parameters
int main(int argc, char* const argv[]) { ... return 0; }
```

The reason for having the parameter option for the main function is to allow input from the command line. When you use the main function with parameters, it saves every group of characters (separated by a space) after the program name as elements in an array named **argv**.

Since the main function has the return type of **int**, the programmer must always have a return statement in the code. The number that is returned is used to inform the calling program what the result of the program's execution was. Returning 0 signals that there were no problems.

## C++ Recursion

When function is called within the same function, it is known as recursion in C++. The function which calls the same function, is known as recursive function.

A function that calls itself, and doesn't perform any task after function call, is known as tail recursion. In tail recursion, we generally call the same function with return statement.

### Syntax:

#### C++

```
recursionfunction()
{
    recursionfunction(); // calling self function
}
```

To know more see [this article](#).

## C++ Passing Array to Function

In C++, to reuse the array logic, we can create function. To pass array to function in C++, we need to provide only array name.

```
functionname(arrayname); //passing array to function
```

### Example: Print minimum number

---

## C++

```
#include <iostream>
using namespace std;
void printMin(int arr[5]);
int main()
{
    int ar[5] = { 30, 10, 20, 40, 50 };
    printMin(ar); // passing array to function
}
void printMin(int arr[5])
{
    int min = arr[0];
    for (int i = 0; i < 5; i++) {
        if (min > arr[i]) {
            min = arr[i];
        }
    }
    cout << "Minimum element is: " << min << "\n";
}

// Code submitted by Susobhan Akhuli
```

## Output

Minimum element is: 10

Time complexity:  $O(n)$  where  $n$  is the size of array

Space complexity:  $O(n)$  where  $n$  is the size of array.

## C++ Overloading (Function)

If we create two or more members having the same name but different in number or type of parameter, it is known as C++ overloading. In C++, we can overload:

- *methods,*

- constructors and
- indexed properties

It is because these members have parameters only.

## Types of overloading in C++ are:

- Function overloading
- Operator overloading

## C++ Function Overloading

Function Overloading is defined as the process of having two or more function with the same name, but different in parameters is known as function overloading in C++. In function overloading, the function is redefined by using either different types of arguments or a different number of arguments. It is only through these differences compiler can differentiate between the functions.

The advantage of Function overloading is that it increases the readability of the program because you don't need to use different names for the same action.

### Example: changing number of arguments of add() method

---

## C++

```
// program of function overloading when number of arguments
// vary
#include <iostream>
using namespace std;
class Cal {
public:
    static int add(int a, int b) { return a + b; }
    static int add(int a, int b, int c)
    {
        return a + b + c;
    }
};
int main(void)
{
    Cal C; // class object declaration.
    cout << C.add(10, 20) << endl;
    cout << C.add(12, 20, 23);
    return 0;
}
// Code Submitted By Susobhan Akhuli
```

## Output

```
30
55
```

**Time complexity:** O(1)

**Space complexity:** O(1)

Example: when the type of the arguments vary.

---

## C++

```
// Program of function overloading with different types of
// arguments.
#include <iostream>
using namespace std;
int mul(int, int);
float mul(float, int);

int mul(int a, int b) { return a * b; }
float mul(double x, int y) { return x * y; }
int main()
{
    int r1 = mul(6, 7);
    float r2 = mul(0.2, 3);
    cout << "r1 is : " << r1 << endl;
    cout << "r2 is : " << r2 << endl;
    return 0;
}

// Code Submitted By Susobhan Akhuli
```

## Output

```
r1 is : 42
r2 is : 0.6
```

**Time Complexity:** O(1)

**Space Complexity:** O(1)

## Function Overloading and Ambiguity

When the compiler is unable to decide which function is to be invoked among the overloaded functions, this situation is known as *function overloading*.

When the compiler shows the ambiguity error, the compiler does not run the program.

### Causes of Function Overloading:

- Type Conversion.
- Function with default arguments.
- Function with pass by reference.

## Type Conversion:-

---

### C++

```
#include <iostream>
using namespace std;
void fun(int);
void fun(float);
void fun(int i) { cout << "Value of i is : " << i << endl; }
void fun(float j)
{
    cout << "Value of j is : " << j << endl;
}
int main()
{
    fun(12);
    fun(1.2);
    return 0;
}

// Code Submitted By Susobhan Akhuli
```

The above example shows an error "*call of overloaded 'fun(double)' is ambiguous*". The `fun(10)` will call the first function. The `fun(1.2)` calls the second function according to our prediction. But, this does not refer to any function as in C++, all the floating point constants are treated as double not as a float. If we replace float to double, the program works. Therefore, this is a type conversion from float to double.

## Function with Default Arguments:-

---

### C++

```
#include <iostream>
using namespace std;
void fun(int);
void fun(int, int);
void fun(int i) { cout << "Value of i is : " << i << endl; }
void fun(int a, int b = 9)
{
    cout << "Value of a is : " << a << endl;
    cout << "Value of b is : " << b << endl;
}
int main()
{
    fun(12);
```

```

    return 0;
}

// Code Submitted By Susobhan Akhuli

```

The above example shows an error "*call of overloaded 'fun(int)' is ambiguous*". The fun(int a, int b=9) can be called in two ways: first is by calling the function with one argument, i.e., fun(12) and another way is calling the function with two arguments, i.e., fun(4,5). The fun(int i) function is invoked with one argument. Therefore, the compiler could not be able to select among fun(int i) and fun(int a,int b=9).

### Function with Pass By Reference:-

---

## C++

```

#include <iostream>
using namespace std;
void fun(int);
void fun(int&);

int main()
{
    int a = 10;
    fun(a); // error, which fun()?
    return 0;
}

void fun(int x) { cout << "Value of x is : " << x << endl; }
void fun(int& b)
{
    cout << "Value of b is : " << b << endl;
}

```

// Code Submitted By Susobhan Akhuli

The above example shows an error "*call of overloaded 'fun(int&)' is ambiguous*". The first function takes one integer argument and the second function takes a reference parameter as an argument. In this case, the compiler does not know which function is needed by the user as there is no syntactical difference between the fun(int) and fun(int &).

### Friend Function

- A friend function is a special function in C++ which in-spite of not being member function of a class has privilege to access private and protected data of a class.
- A friend function is a non member function or ordinary function of a class, which is declared as a friend using the keyword "friend" inside the class. By declaring a function as a friend, all the access permissions are given to the function.

- The keyword "friend" is placed only in the function declaration of the friend function and not in the function definition.
- When friend function is called neither name of object nor dot operator is used. However it may accept the object as argument whose value it wants to access.
- Friend function can be declared in any section of the class i.e. public or private or protected.

## Declaration of friend function in C++

Syntax :

```
class <class_name>
{
    friend <return_type> <function_name>(argument/s);
};
```

Example\_1: Find the largest of two numbers using Friend Function

---

## C++

```
#include<iostream>
using namespace std;
class Largest
{
    int a,b,m;
    public:
        void set_data();
        friend void find_max(Largest);
};

void Largest::set_data()
{
    cout<<"Enter the First No:";
    cin>>a;
    cout<<"Enter the Second No:";
    cin>>b;
}

void find_max(Largest t)
{
    if(t.a>t.b)
        t.m=t.a;
    else
        t.m=t.b;

    cout<<"Maximum Number is\t"<<t.m;
}

main()
{
    Largest l;
```

```
1.set_data();  
find_max(1);  
return 0;  
}
```

## Output

```
Enter the First No:Enter the Second No:Maximum Number is 2117529904
```

```
461
```

## Related Articles

1. [Static functions in C](#)
2. [Write one line functions for strcat\(\) and strcmp\(\)](#)
3. [Can Static Functions Be Virtual in C++?](#)
4. [Functions that cannot be overloaded in C++](#)
5. [Functions that are executed before and after main\(\) in C](#)
6. [Pure Functions](#)
7. [Can Virtual Functions be Private in C++?](#)
8. [Virtual Functions and Runtime Polymorphism in C++](#)
9. [Can Virtual Functions be Inlined in C++?](#)
10. [Macros vs Functions](#)

[Previous](#)[Next](#)

## Article Contributed By :



GeeksforGeeks

**SALE!**  
**GeeksforGeeks Courses Upto 25% Off Enroll Now!**



Save 25% on Courses DSA Data Structures Algorithms Interview Preparation Data Science T

# Default Arguments in C++

Difficulty Level : Easy • Last Updated : 27 Jan, 2023

[Read](#) [Discuss](#) [Courses](#) [Practice](#) [Video](#)

A default argument is a value provided in a function declaration that is automatically assigned by the compiler if the calling function doesn't provide a value for the argument. In case any value is passed, the default value is overridden.

**1)** The following is a simple C++ example to demonstrate the use of default arguments. Here, we don't have to write 3 sum functions; only one function works by using the default values for 3rd and 4th arguments.

## CPP

```
// CPP Program to demonstrate Default Arguments
#include <iostream>
using namespace std;

// A function with default arguments,
// it can be called with
// 2 arguments or 3 arguments or 4 arguments.
int sum(int x, int y, int z = 0, int w = 0) //assigning default values to z,w as 0
{
    return (x + y + z + w);
}

// Driver Code
int main()
{
    // Statement 1
    cout << sum(10, 15) << endl;

    // Statement 2
    cout << sum(10, 15, 25) << endl;

    // Statement 3
    cout << sum(10, 15, 25, 30) << endl;
```

```

    return 0;
}

```

## Output

```

25
50
80

```

### Time Complexity: O(1)

AD

### Space Complexity: O(1)

**Explanation:** In statement 1, only two values are passed, hence the variables z and w take the default values of 0. In statement 2, three values are passed, so the value of z is overridden with 25. In statement 3, four values are passed, so the values of z and w are overridden with 25 and 30 respectively.

**2)** If function overloading is done containing the default arguments, then we need to make sure it is not ambiguous to the compiler, otherwise it will throw an error. The following is the modified version of the above program:

## CPP

```

// CPP Program to demonstrate Function overloading in
// Default Arguments
#include <iostream>
using namespace std;

// A function with default arguments, it can be called with
// 2 arguments or 3 arguments or 4 arguments.
int sum(int x, int y, int z = 0, int w = 0)
{
    return (x + y + z + w);
}
int sum(int x, int y, float z = 0, float w = 0)
{
    return (x + y + z + w);
}
// Driver Code

```

```

int main()
{
    cout << sum(10, 15) << endl;
    cout << sum(10, 15, 25) << endl;
    cout << sum(10, 15, 25, 30) << endl;
    return 0;
}

```

**Error:**

```

prog.cpp: In function 'int main()':
prog.cpp:17:20: error: call of overloaded
'sum(int, int)' is ambiguous
    cout << sum(10, 15) << endl;
                           ^
prog.cpp:6:5: note: candidate:
int sum(int, int, int)
int sum(int x, int y, int z=0, int w=0)
^
prog.cpp:10:5: note: candidate:
int sum(int, int, float, float)
int sum(int x, int y, float z=0, float w=0)
^

```

- 3)** A constructor can contain default parameters as well. A default constructor can either have no parameters or parameters with default arguments.

**C++**

```

// CPP code to demonstrate use of default arguments in
// Constructors

#include <iostream>
using namespace std;

class A {
private:
    int var = 0;
public:
    A(int x = 0): var(x){} // default constructor with one argument
                           // Note that var(x) is the syntax in c++ to do : "va
    void setVar(int s){
        var = s; // OR => this->var = s;
        return;
    }
    int getVar(){
        return var; // OR => return this->var;
    }
};

```

```

int main(){
    A a(1);

    a.setVar(2);

    cout << "var = " << a.getVar() << endl;

    /* ANOTHER APPROACH:
    A *a = new A(1);

    a->setVar(2);

    cout << "var = " << a->getVar() << endl;
    */
}

// contributed by Francisco Vargas #pt

```



**Explanation:** Here, we see a default constructor with no arguments and a default constructor with one default argument. The default constructor with argument has a default parameter x, which has been assigned a value of 0.

### Key Points:

- Default arguments are different from constant arguments as constant arguments can't be changed whereas default arguments can be overwritten if required.
- Default arguments are overwritten when the calling function provides values for them. For example, calling the function sum(10, 15, 25, 30) overwrites the values of z and w to 25 and 30 respectively.
- When a function is called, the arguments are copied from the calling function to the called function in the order left to right. Therefore, sum(10, 15, 25) will assign 10, 15, and 25 to x, y, and z respectively, which means that only the default value of w is used.
- Once a default value is used for an argument in the function definition, all subsequent arguments to it must have a default value as well. It can also be stated that the default arguments are assigned from right to left. For example, the following function definition is invalid as the subsequent argument of the default variable z is not default.

```

// Invalid because z has default value, but w after it doesn't have a
default value
int sum(int x, int y, int z = 0, int w).

```

### Advantages of Default Arguments:

- Default arguments are useful when we want to increase the capabilities of an existing function as we can do it just by adding another default argument to the function.
- It helps in reducing the size of a program.

- It provides a simple and effective programming approach.
- Default arguments improve the consistency of a program.

## Disadvantages of Default Arguments:

- It increases the execution time as the compiler needs to replace the omitted arguments by their default values in the function call.

203

## Related Articles

1. [Templates and Default Arguments](#)
2. [Some Interesting facts about default arguments in C++](#)
3. [Default Arguments and Virtual Function in C++](#)
4. [When do we pass arguments by reference or pointer?](#)
5. [Template non-type arguments in C++](#)
6. [Command Line Arguments in C/C++](#)
7. [Does C++ compiler create default constructor when we write our own?](#)
8. [Default constructor in Java](#)
9. [C++ default constructor | Built-in types for int\(\), float, double\(\)](#)
10. [Default Assignment Operator and References in C++](#)

Previous

Next

## Article Contributed By :



GeeksforGeeks

## Vote for difficulty

**SALE!**  
**GeeksforGeeks Courses Upto 25% Off Enroll Now!**



Save 25% on Courses DSA Data Structures Algorithms Interview Preparation Data Science T

# Inline Functions in C++

Difficulty Level : Medium • Last Updated : 05 Mar, 2023

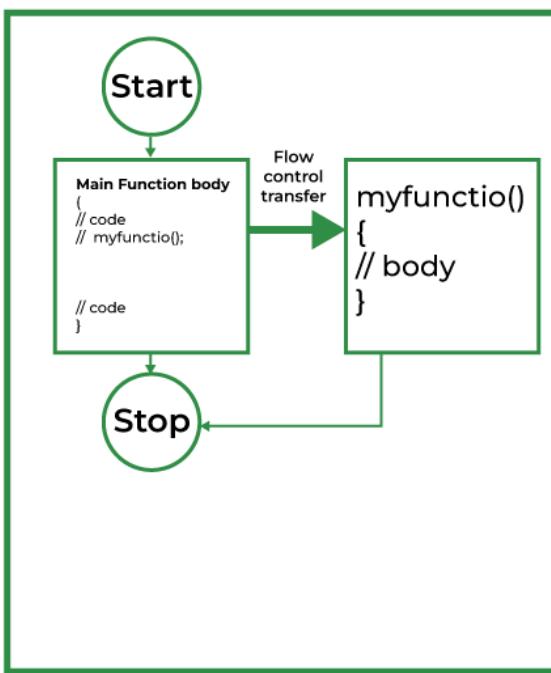
[Read](#) [Discuss\(20\)](#) [Courses](#) [Practice](#) [Video](#)

C++ provides inline functions to reduce the function call overhead. An 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 the inline function call. This substitution is performed by the C++ compiler at compile time. An inline function may increase efficiency if it is small.

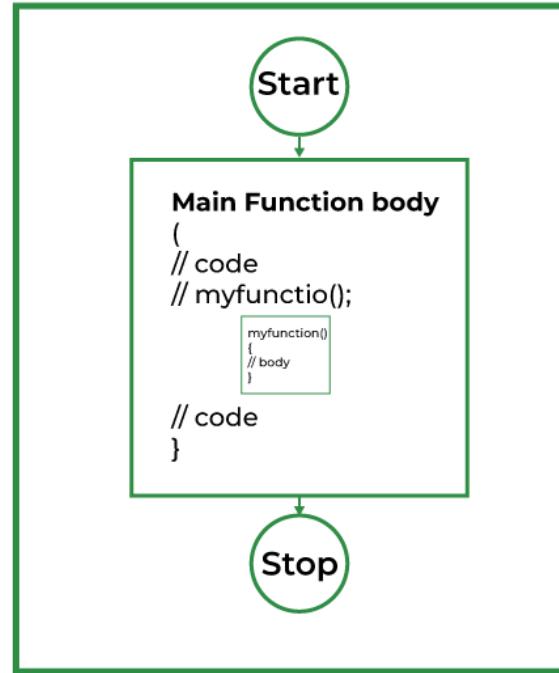
## Syntax:

```
inline return-type function-name(parameters)
{
    // function code
}
```

## Normal Function



## Inline Function



Remember, inlining is only a request to the compiler, not a command. The compiler can ignore the request for inlining.

**The compiler may not perform inlining in such circumstances as:**

AD

1. If a function contains a loop. (*for, while and 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 a function body.
5. If a function contains a switch or goto statement.

## Why Inline Functions are Used?

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 the function is less than the switching time from the caller function to called function (callee).

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 the execution time of a small function is less than the switching time.

## Inline functions Advantages:

1. Function call overhead doesn't occur.
2. It also saves the overhead of push/pop variables on the stack when a function is called.
3. It also saves the overhead of a return call from a function.
4. When you inline a function, you may enable the compiler to perform context-specific optimization on the body of the function. Such optimizations are not possible for normal function calls. Other optimizations can be obtained by considering the flows of the calling context and the called context.
5. An inline function may be useful (if it is small) for embedded systems because inline can yield less code than the function called preamble and return.

## Inline function Disadvantages:

1. The added variables from the inlined function consume additional registers, After the in-lining function if the variable number which is going to use the register increases then they may create overhead on register variable resource utilization. This means that when the inline function body is substituted at the point of the function call, the total number of variables used by the function also gets inserted. So the number of registers 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 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 the 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. The 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 the compiler would be required 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 the size of the binary executable file. Thrashing in memory causes the performance of the computer to degrade. The following program demonstrates the use of the inline function.

### Example:

---

### C++

```
#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

## Inline function and classes

It is also possible to define the inline function inside the class. In fact, all the functions defined inside the class are implicitly inline. Thus, all the restrictions of inline functions are also applied here. If you need to explicitly declare an inline function in the class then just declare the function inside the class and define it outside the class using the inline keyword.

### Syntax:

```
class S
{
public:
    inline int square(int s) // redundant use of inline
    {
        // this function is automatically inline
        // function body
    }
}
```

```
    }  
};
```

The above style is considered a bad programming style. The best programming style is to just write the prototype of the function inside the class and specify it as an inline in the function definition.

### For Example:

```
class S  
{  
public:  
    int square(int s); // declare the function  
};  
  
inline int S::square(int s) // use inline prefix  
{  
}
```

### Example:

---

## C++

```
// C++ Program to demonstrate inline functions and classes  
#include <iostream>  
  
using namespace std;  
  
class operation {  
    int a, b, add, sub, mul;  
    float div;  
  
public:  
    void get();  
    void sum();  
    void difference();  
    void product();  
    void division();  
};  
inline void operation ::get()  
{  
    cout << "Enter first value:";  
    cin >> a;  
    cout << "Enter second value:";  
    cin >> b;  
}  
  
inline void operation ::sum()  
{
```

```

    add = a + b;
    cout << "Addition of two numbers: " << a + b << "\n";
}

inline void operation ::difference()
{
    sub = a - b;
    cout << "Difference of two numbers: " << a - b << "\n";
}

inline void operation ::product()
{
    mul = a * b;
    cout << "Product of two numbers: " << a * b << "\n";
}

inline void operation ::division()
{
    div = a / b;
    cout << "Division of two numbers: " << a / b << "\n";
}

int main()
{
    cout << "Program using inline function\n";
    operation s;
    s.get();
    s.sum();
    s.difference();
    s.product();
    s.division();
    return 0;
}

```

## Output:

```

Enter first value: 45
Enter second value: 15
Addition of two numbers: 60
Difference of two numbers: 30
Product of two numbers: 675
Division of two numbers: 3

```

## What is wrong with the macro?

Readers familiar with the C language know that the C language uses macro. The preprocessor replaces all macro calls directly within the macro code. It is recommended to always use the inline function instead of the macro. According to Dr. Bjarne Stroustrup, the creator of C++ 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 the class. Macros look like function calls but they are actually not.

### **Example:**

---

## C++

```
// C++ Program to demonstrate working of macro
#include <iostream>
using namespace std;
class S {
    int m;

public:
    // error
#define MAC(S)::m
};
```

### **Output:**

```
Error: ":" may not appear in macro parameter list
#define MAC(S)::m
```

C++ compiler checks the argument types of inline functions and necessary conversions are performed correctly. The preprocessor macro is not capable of doing this. One other thing is that the macros are managed by the preprocessor and inline functions are managed by the C++ compiler. Remember: It is true that all the functions defined inside the class are implicitly inline and the C++ compiler will perform inline calls of these functions, but the C++ compiler cannot perform inline if the function is virtual. The reason is called to a virtual function is resolved at runtime instead of compile-time. Virtual means waiting until runtime and inline means during compilation, if the compiler doesn't know which function will be called, how it can perform inlining? One other thing to remember is that it is only useful to make the function inline if the time spent during a function call is more compared to the function body execution time.

### **An example where the inline function has no effect at all:**

```
inline void show()
{
    cout << "value of S = " << S << endl;
}
```

The above function relatively takes a long time to execute. In general, a function that performs an input-output (I/O) operation shouldn't be defined as inline because it spends a

considerable amount of time. Technically inlining of the `show()` function is of limited value because the amount of time the I/O statement will take far exceeds the overhead of a function call. Depending upon the compiler you are using the compiler may show you a warning if the function is not expanded inline.

Programming languages like Java & C# don't support inline functions. But in Java, the compiler can perform inlining when the small final method is called because final methods can't be overridden by subclasses, and the call to a final method is resolved at compile time.

In C# JIT compiler can also optimize code by inlining small function calls (like replacing the body of a small function when it is called in a loop). The last thing to keep in mind is that inline functions are a valuable feature of C++. Appropriate use of inline functions can provide performance enhancement but if inline functions are used arbitrarily then they can't provide better results. In other words, don't expect a better performance of the program. Don't make every function inline. It is better to keep inline functions as small as possible.

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

338

## Related Articles

1. Mathematical Functions in Python | Set 1 (Numeric Functions)

---

2. Mathematical Functions in Python | Set 2 (Logarithmic and Power Functions)

---

3. Difference between Inline and Macro in C++

---

4. Difference Between Inline and Normal Function in C++

---

5. Difference between virtual function and inline function in C++

---

6. C++ Inline Namespaces and Usage of the "using" Directive Inside Namespaces

---

7. Can Static Functions Be Virtual in C++?

---

8. Virtual Functions in Derived Classes in C++

---

9. Functions that cannot be overloaded in C++



SALE!  
GeeksforGeeks Courses Upto 25% Off Enroll Now!



Save 25% on Courses DSA Data Structures Algorithms Interview Preparation Data Science T

# Return From Void Functions in C++

Difficulty Level : Easy • Last Updated : 27 Jan, 2023

[Read](#) [Discuss](#) [Courses](#) [Practice](#) [Video](#)

Void functions are known as **Non-Value Returning functions**. They are "void" due to the fact that they are not supposed to return values. True, but not completely. We cannot return values but there is something we can surely return from void functions. **Void functions do not have a return type, but they can do return values.** Some of the cases are listed below:

**1) A Void Function Can Return:** We can simply write a return statement in a void fun(). In fact, it is considered a good practice (for readability of code) to write a return; statement to indicate the end of the function.

## CPP

```
// CPP Program to demonstrate void functions
#include <iostream>
using namespace std;

void fun()
{
    cout << "Hello";

    // We can write return in void
    return;
}

// Driver Code
int main()
{
    fun();
    return 0;
}
```

## Output

Hello

**Time Complexity:** O(1)

**Space Complexity:** O(1)

AD

**2) A void fun() can return another void function:** A void function can also call another void function while it is terminating. For example,

---

## CPP

```
// C++ code to demonstrate void()
// returning void()
#include <iostream>
using namespace std;

// A sample void function
void work()
{
    cout << "The void function has returned "
        " a void() !!! \n";
}

// Driver void() returning void work()
void test()
{
    // Returning void function
    return work();
}

// Driver Code
int main()
{
    // Calling void function
    test();
    return 0;
}
```

## Output

The void function has returned a void() !!!

**Time Complexity:** O(1)

**Space Complexity:** O(1)

The above code explains how void() can actually be useful to return void functions without giving errors.

**3) A void() can return a void value:** A void() cannot return a value that can be used. But it can return a value that is void without giving an error. For example,

## CPP

```
// C++ code to demonstrate void()
// returning a void value
#include <iostream>
using namespace std;

// Driver void() returning a void value
void test()
{
    cout << "Hello";

    // Returning a void value
    return (void)"Doesn't Print";
}

// Driver Code
int main()
{
    test();
    return 0;
}
```

## Output

Hello

**Time Complexity:** O(1)

**Space Complexity:** O(1)

This article is contributed by **Manjeet Singh**. If you like GeeksforGeeks and would like to contribute, you can also write an article using [write.geeksforgeeks.org](https://write.geeksforgeeks.org) or mail your article to review-team@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

**SALE!**  
**GeeksforGeeks Courses Upto 25% Off Enroll Now!**



Save 25% on Courses DSA Data Structures Algorithms Interview Preparation Data Science T

# Functors in C++

Difficulty Level : Medium • Last Updated : 27 Jan, 2023

[Read](#) [Discuss](#) [Courses](#) [Practice](#) [Video](#)

Please note that the title is **Functors** (Not Functions)!!

Consider a function that takes only one argument. However, while calling this function we have a lot more information that we would like to pass to this function, but we cannot as it accepts only one parameter. What can be done?

One obvious answer might be global variables. However, good coding practices do not advocate the use of global variables and say they must be used only when there is no other alternative.

**Functors** are objects that can be treated as though they are a function or function pointer. Functors are most commonly used along with STLs in a scenario like following:

AD

Below program uses [transform\(\) in STL](#) to add 1 to all elements of arr[].

## CPP

```
// A C++ program uses transform() in STL to add
// 1 to all elements of arr[]
#include <bits/stdc++.h>
using namespace std;
```

```

int increment(int x) { return (x+1); }

int main()
{
    int arr[] = {1, 2, 3, 4, 5};
    int n = sizeof(arr)/sizeof(arr[0]);

    // Apply increment to all elements of
    // arr[] and store the modified elements
    // back in arr[]
    transform(arr, arr+n, arr, increment);

    for (int i=0; i<n; i++)
        cout << arr[i] << " ";

    return 0;
}

```

Output:

2 3 4 5 6

**Time Complexity:** O(n)

**Space Complexity:** O(n) where n is the size of the array.

This code snippet adds only one value to the contents of the arr[]. Now suppose, that we want to add 5 to contents of arr[].

See what's happening? As transform requires a unary function(a function taking only one argument) for an array, we cannot pass a number to increment(). And this would, in effect, make us write several different functions to add each number. What a mess. This is where functors come into use.

A functor (or function object) is a C++ class that acts like a function. Functors are called using the same old function call syntax. To create a functor, we create a object that overloads the *operator()*.

The line,

MyFunctor(10);

Is same as

MyFunctor.operator()(10);

Let's delve deeper and understand how this can actually be used in conjunction with STLs.

## CPP

```
// C++ program to demonstrate working of
// functors.
#include <bits/stdc++.h>
using namespace std;

// A Functor
class increment
{
private:
    int num;
public:
    increment(int n) : num(n) { }

    // This operator overloading enables calling
    // operator function () on objects of increment
    int operator () (int arr_num) const {
        return num + arr_num;
    }
};

// Driver code
int main()
{
    int arr[] = {1, 2, 3, 4, 5};
    int n = sizeof(arr)/sizeof(arr[0]);
    int to_add = 5;

    transform(arr, arr+n, arr, increment(to_add));

    for (int i=0; i<n; i++)
        cout << arr[i] << " ";
}
```

Output:

6 7 8 9 10

**Time Complexity:** O(n)

**Space Complexity:** O(n) where n is the size of the array.

Thus, here, Increment is a functor, a c++ class that acts as a function.

The line,

```
transform(arr, arr+n, arr, increment(to_add));
```

is the same as writing below two lines,

```
// Creating object of increment
increment obj(to_add);
```

```
// Calling () on object  
transform(arr, arr+n, arr, obj);
```

Thus, an object *a* is created that overloads the *operator()*. Hence, functors can be used effectively in conjunction with C++ STLs. This article is contributed by **Supriya Srivatsa**. If you like GeeksforGeeks and would like to contribute, you can also write an article and mail your article to review-team@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

185

## Related Articles

1. [Sort an array according to absolute difference with given value using Functors](#)

---
2. [C++ - Difference Between Functors and Functions](#)

---
3. [How to Use Binder and Bind2nd Functors in C++ STL?](#)

---
4. [main Function in C](#)

---
5. [printf in C](#)

---
6. [C Program to Implement Max Heap](#)

---
7. [C Program to Implement Min Heap](#)

---
8. [C++ Error - Does not name a type](#)

---
9. [Execution Policy of STL Algorithms in Modern C++](#)

---
10. [C++ Program To Print Pyramid Patterns](#)

---

Previous

Next

## Article Contributed By :



GeeksforGeeks

SALE!  
GeeksforGeeks Courses Upto 25% Off Enroll Now!



Save 25% on Courses DSA Data Structures Algorithms Interview Preparation Data Science T

# C Pointers

Difficulty Level : Easy • Last Updated : 23 Mar, 2023

[Read](#) [Discuss\(30+\)](#) [Courses](#) [Practice](#) [Video](#)

Pointers in C are used to store the address of variables or a memory location. This variable can be of any data type i.e, int, char, function, array, or any other pointer. Pointers are one of the core concepts of C programming language that provides low-level memory access and facilitates dynamic memory allocation.

## What is a Pointer in C?

A pointer is a derived data type in C that can store the address of other variables or a memory. We can access and manipulate the data stored in that memory location using pointers.

## Syntax of C Pointers

```
datatype * pointer_name;
```

The above syntax is the generic syntax of C pointers. The actual syntax depends on the type of data the pointer is pointing to.

## How to Use Pointers?

To use pointers in C, we must understand below two operators:

AD

## 1. Addressof Operator

The addressof operator ( & ) is a unary operator that returns the address of its operand. Its operand can be a variable, function, array, structure, etc.

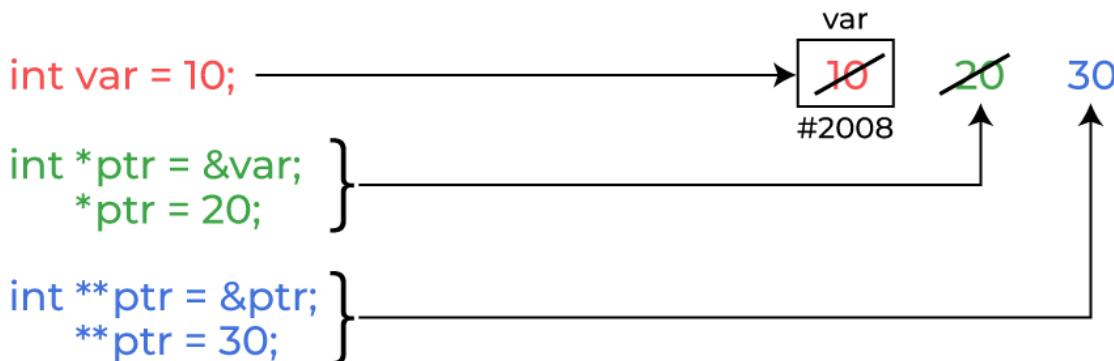
### Syntax of Address of Operator

```
&variable_name;
```

## 2. Dereferencing Operator

The dereference operator ( \* ), also known as the indirection operator is a unary operator. It is used in pointer declaration and dereferencing.

## How pointer works in C



## C Pointer Declaration

In pointer declaration, we only declare the pointer but do not initialize it. To declare a pointer, we use the \* dereference operator before its name.

```
data_type * pointer_name;
```

The pointer declared here will point to some random memory address as it is not initialized. Such pointers are also called wild pointers that we will study later in this article.

## C Pointer Initialization

When we assign some value to the pointer, it is called Pointer Initialization in C. There are two ways in which we can initialize a pointer in C of which the first one is:

### Method 1: C Pointer Definition

```
datatype * pointer_name = address;
```

The above method is called Pointer Definition as the pointer is declared and initialized at the same time.

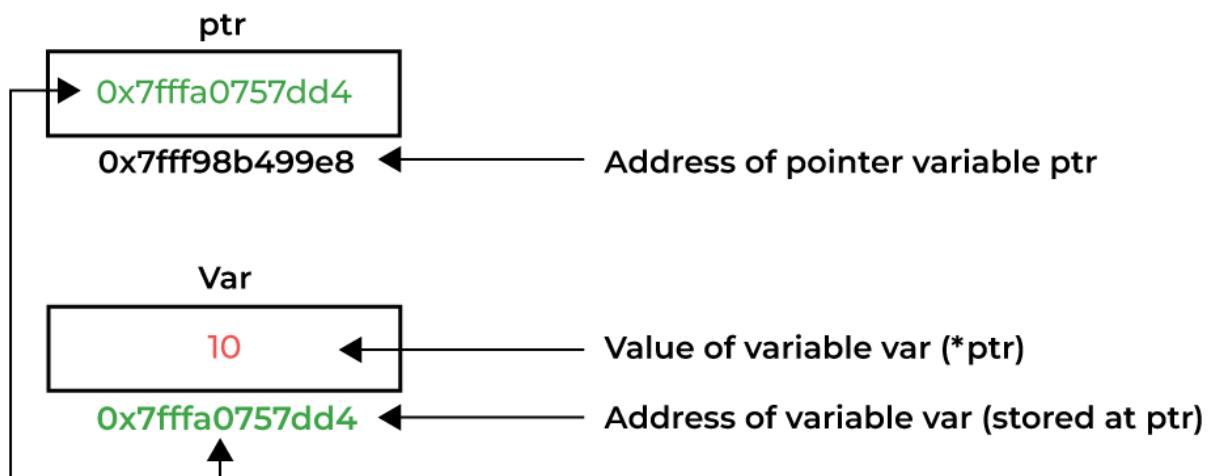
## Method 2: Initialization After Declaration

The second method of pointer initialization in C is the assigning some address after the declaration.

```
datatype * pointer_name;
pointer_name = address;
```

## Dereferencing a C Pointer

Dereferencing is the process of accessing the value stored in the memory address specified in the pointer. We use dereferencing operator for that purpose.



Dereferencing a Pointer in C

## Example: C Program to demonstrate how to use pointers in C.

### C

```
// C program to illustrate Pointers
#include <stdio.h>

void geeks()
{
    int var = 20;

    // declare pointer variable
```

```

int* ptr;

// note that data type of ptr and var must be same
ptr = &var;

// assign the address of a variable to a pointer
printf("Value at ptr = %p \n", ptr);
printf("Value at var = %d \n", var);
printf("Value at *ptr = %d \n", *ptr);

}

// Driver program
int main()
{
    geeks();
    return 0;
}

```

## Output

```

Value at ptr = 0x7ffd15b5deec
Value at var = 20
Value at *ptr = 20

```

## Types of Pointers

Pointers can be classified into many different types based on the parameter on which we are defining their types. If we consider the type of variable stored in the memory location pointed by the pointer, then the pointers can be classified into the following types:

### 1. Integer Pointers

As the name suggests, these are the pointers that point to the integer values.

#### Syntax of Integer Pointers

```
int *pointer_name;
```

These pointers are pronounced as Pointer to Integer.

Similarly, a pointer can point to any primitive data type. The syntax will change accordingly. It can point also point to derived data types such as arrays and user-defined data types such as structures.

### 2. Array Pointer

Pointers and Array are closely related to each other. Even the array name is the pointer to its first element. They are also known as [Pointer to Arrays](#). We can create a pointer to an

array using the given syntax.

### Syntax of Array Pointers

```
char *pointer_name = &array_name;
```

Pointer to Arrays exhibits some interesting properties which we discussed later in this article.

## 3. Structure Pointer

The pointer pointing to the structure type is called [Structure Pointer](#) or Pointer to Structure. It can be declared in the same way as we declare the other primitive data types.

### Syntax of Structure Pointer

```
struct struct_name *pointer_name;
```

## 4. Function Pointers

Function pointers point to the functions. They are different from the rest of the pointers in the sense that instead of pointing to the data, they point to the code. Let's consider a function prototype – **int func (int, char)**, the [function pointer](#) for this function will be

### Syntax of Function Pointer

```
int (*pointer_name)(int, int);
```

Keep in mind that the syntax of the function pointers changes according to the function prototype.

## 5. Double Pointers

In C language, we can define a pointer that stores the memory address of another pointer. Such pointers are called double-pointers or [pointers-to-pointer](#). Instead of pointing to a data value, they point to another pointer.

### Syntax of Double Pointer in C

```
datatype ** pointer_name;
```

### Dereferencing in Double Pointer

```
*pointer_name; // get the address stored in the inner level pointer  
**pointer_name; // get the value pointed by inner level pointer
```

**Note:** In C, we can create multi-level pointers with any number of levels such as - \*\*\*ptr3, \*\*\*\*ptr4, \*\*\*\*\*ptr5 and so on.

## 6. NULL Pointer

The Null Pointers are those pointers that do not point to any memory location. They can be created by assigning a NULL value to the pointer. A pointer of any type can be assigned the NULL value.

### Syntax of NULL Pointer in C

```
data_type *pointer_name = NULL;  
or  
pointer_name = NULL
```

It is said to be good practice to assign NULL to the pointers currently not in use.

## 7. Void Pointer

The Void pointers in C are the pointers of type void. It means that they do not have any associated data type. They are also called **generic pointers** as they can point to any type and can be typecasted to any type.

### Syntax of Void Pointer

```
void * pointer_name;
```

One of the main properties of void pointers is that they cannot be dereferenced.

## 8. Wild Pointers

The Wild Pointers are pointers that have not been initialized with something yet. These types of C-pointers can cause problems in our programs and can eventually cause them to crash.

### Example of Wild Pointers

```
int *ptr;  
char *str;
```

## 9. Constant Pointers

In constant pointers, the memory address stored inside the pointer is constant and cannot be modified once it is defined. It will always point to the same memory address.

### Syntax of Constant Pointer

```
const data_type * pointer_name;
```

## 10. Pointer to Constant

The pointers pointing to a constant value that cannot be modified are called pointers to a constant. Here we can only access the data pointed by the pointer, but cannot modify it. Although, we can change the address stored in the pointer to constant.

### Syntax to Pointer to Constant

```
data_type * const pointer_name;
```

## Other Types of Pointers in C:

There are also the following types of pointers available to use in C apart from those specified above:

- **Far pointer:** A far pointer is typically 32-bit that can access memory outside the current segment.
- **Dangling pointer:** A pointer pointing to a memory location that has been deleted (or freed) is called a dangling pointer.
- **Huge pointer:** A huge pointer is 32-bit long containing segment address and offset address.
- **Complex pointer:** Pointers with multiple levels of indirection.
- **Near pointer:** Near pointer is used to store 16-bit addresses means within the current segment on a 16-bit machine.
- **Normalized pointer:** It is a 32-bit pointer, which has as much of its value in the segment register as possible.
- **File Pointer:** The pointer to a FILE data type is called a stream pointer or a file pointer.

## Size of Pointers in C

The size of the pointers in C is equal for every pointer type. The size of the pointer does not depend on the type it is pointing to. It only depends on the operating system and CPU architecture. The size of pointers in C is

- **8 bytes for a 64-bit System**
- **4 bytes for a 32-bit System**

The reason for the same size is that the pointers store the memory addresses, no matter what type they are. As the space required to store the addresses of the different memory locations is the same, the memory required by one pointer type will be equal to the memory required by other pointer types.

## How to find the size of pointers in C?

We can find the size of pointers using the [sizeof operator](#) as shown in the following program:

### Example: C Program to find the size of different pointer types.

---

## C

```
// C Program to find the size of different pointers types
#include <stdio.h>

// dummy structure
struct str {
};

// dummy function
void func(int a, int b){};

int main()
{
    // dummy variables definitions
    int a = 10;
    char c = 'G';
    struct str x;

    // pointer definitions of different types
    int* ptr_int = &a;
    char* ptr_char = &c;
    struct str* ptr_str = &x;
    void (*ptr_func)(int, int) = &func;
    void* ptr_vn = NULL;

    // printing sizes
    printf("Size of Integer Pointer \t:\t%d bytes\n",
           sizeof(ptr_int));
    printf("Size of Character Pointer\t:\t%d bytes\n",
           sizeof(ptr_char));
    printf("Size of Structure Pointer\t:\t%d bytes\n",
           sizeof(ptr_str));
    printf("Size of Function Pointer\t:\t%d bytes\n",
           sizeof(ptr_func));
    printf("Size of NULL Void Pointer\t:\t%d bytes",
           sizeof(ptr_vn));

    return 0;
}
```

## Output

```
Size of Integer Pointer      : 8 bytes
Size of Character Pointer   : 8 bytes
Size of Structure Pointer   : 8 bytes
Size of Function Pointer    : 8 bytes
Size of NULL Void Pointer   : 8 bytes
```

As we can see, no matter what the type of pointer it is, the size of each and every pointer is the same.

Now, one may wonder that if the size of all the pointers is the same, then why do we need to declare the pointer type in the declaration? The type declaration is needed in the pointer for dereferencing and pointer arithmetic purposes.

## Pointer Arithmetic

Only a limited set of operations can be performed on pointers. The [Pointer Arithmetic](#) refers to the legal or valid operations that can be performed on a pointer. It is slightly different from the ones that we generally use for mathematical calculations. The operations are:

- Increment in a Pointer
- Decrement in a Pointer
- Addition of integer to a pointer
- Subtraction of integer to a pointer
- Subtracting two pointers of the same type
- Comparison of pointers of the same type.
- Assignment of pointers of the same type.

## C

```
// C program to illustrate Pointer Arithmetic

#include <stdio.h>

int main()
{
    // Declare an array
    int v[3] = { 10, 100, 200 };

    // Declare pointer variable
    int* ptr;

    // Assign the address of v[0] to ptr
    ptr = v;
```

```

for (int i = 0; i < 3; i++) {

    // print value at address which is stored in ptr
    printf("Value of *ptr = %d\n", *ptr);

    // print value of ptr
    printf("Value of ptr = %p\n\n", ptr);

    // Increment pointer ptr by 1
    ptr++;
}
return 0;
}

```

## Output

Value of \*ptr = 10  
 Value of ptr = 0x7ffe8ba7ec50

Value of \*ptr = 100  
 Value of ptr = 0x7ffe8ba7ec54

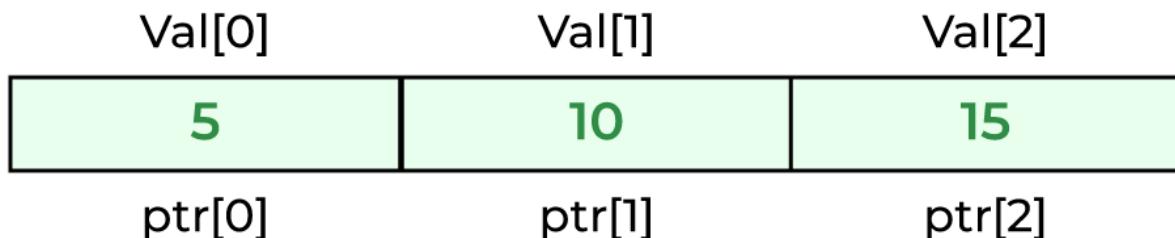
Value of \*ptr = 200  
 Value of ptr = 0x7ffe8ba7ec58

## C Pointers and Arrays Relation

In C programming language, pointers and arrays are closely related. An array name acts like a pointer constant. The value of this pointer constant is the address of the first element. For example, if we have an array named val then **val** and **&val[0]** can be used interchangeably.

If we assign this value to a non-constant pointer to the array, then we can access the elements of the array using this pointer.

### Example 1: Accessing Array Elements using Pointer with Array Subscript



## C

```
// C Program to access array elements using pointer
#include <stdio.h>

void geeks()
{
    // Declare an array
    int val[3] = { 5, 10, 15 };

    // Declare pointer variable
    int* ptr;

    // Assign address of val[0] to ptr.
    // We can use ptr=&val[0];(both are same)
    ptr = val;

    printf("Elements of the array are: ");
    printf("%d, %d, %d", ptr[0], ptr[1], ptr[2]);

    return;
}

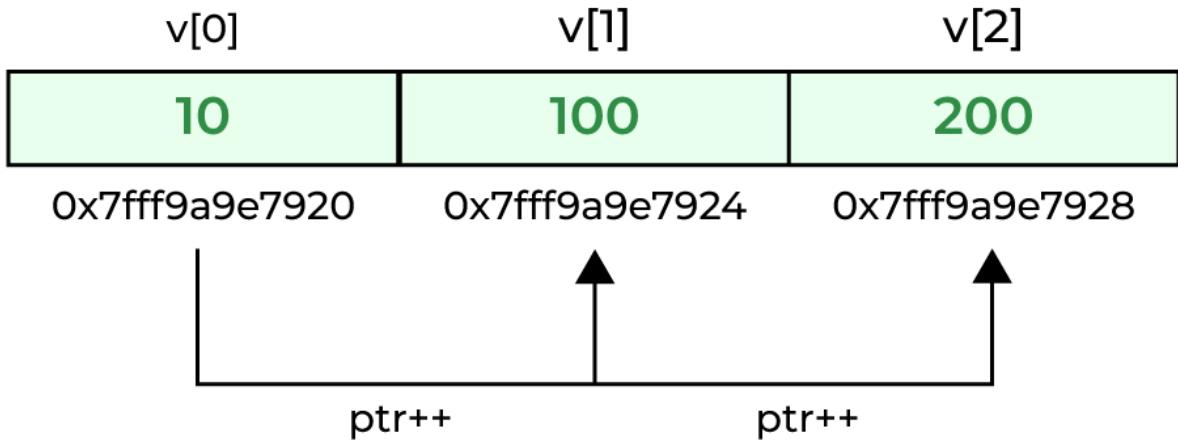
// Driver program
int main()
{
    geeks();
    return 0;
}
```

## Output

Elements of the array are: 5 10 15

Not only that, as the array elements are stored continuously, we can perform arithmetic operations such as increment, decrement, addition, and subtraction of integers on pointer to move between array elements.

## Example 2: Accessing Array Elements using Pointer Arithmetic



## C

```
// C Program to access array elements using pointers
#include <stdio.h>

int main()
{
    // defining array
    int arr[5] = { 1, 2, 3, 4, 5 };

    // defining the pointer to array
    int* ptr_arr = &arr;

    // traversing array using pointer arithmetic
    for (int i = 0; i < 5; i++) {
        printf("%d ", *ptr_arr++);
    }
    return 0;
}
```

## Output

1 2 3 4 5

This concept is not limited to the one-dimensional array, we can refer to a multidimensional array element perfectly fine using this concept.

To know more about pointers to an array, refer to this article – [Pointer to an Array](#).

## Uses of Pointers

The C pointer is a very powerful tool that is widely used in C programming to perform various useful operations. It finds its use in operations such as

1. Pass Arguments by Reference
2. Accessing Array Elements
3. [Return Multiple Values from Function](#)
4. [Dynamic Memory Allocation](#)
5. [Implementing Data Structures](#)
6. In System-Level Programming where memory addresses are useful.
7. In locating the exact value at some memory location.
8. To avoid compiler confusion for the same variable name.
9. To use in Control Tables.

## Advantages of Pointers

- Pointers are used for dynamic memory allocation and deallocation.
- An Array or a structure can be accessed efficiently with pointers
- Pointers are useful for accessing memory locations.
- Pointers are used to form complex data structures such as linked lists, graphs, trees, etc.
- Pointers reduce the length of the program and its execution time as well.

## Disadvantages of Pointers

- Memory corruption can occur if an incorrect value is provided to pointers.
- Pointers are a little bit complex to understand.
- Pointers are majorly responsible for [memory leaks in C](#).
- Pointers are comparatively slower than variables in C.
- Uninitialized pointers might cause a segmentation fault.

## Conclusion

In conclusion, pointers in C are very capable tools and provide C language with its distinguishing features, such as low-level memory access, referencing, etc. But as powerful as they are, they should be used with responsibility as they are one of the most vulnerable parts of the language.

## FAQs on Pointers in C

### 1. Define pointers.

Pointers are the variables that can store the memory address of another variable.

## 2. What is the difference between a constant pointer and a pointer to a constant?

A constant pointer points to the fixed memory location, i.e. we cannot change the memory address stored inside the constant pointer.

On the other hand, the pointer to a constant point to the memory with a constant value.

## 3. What is pointer to pointer?

A pointer to a pointer (also known as a double pointer) stores the address of another pointer.

## 4. Does pointer size depends on its type?

No, the pointer size does not depend upon its type. It only depends on the operating system and CPU architecture.

## 5. What are the differences between an array and a pointer?

The following table list the [differences between an array and a pointer](#):

Pointer	Array
A pointer is a derived data type that can store the address of other variables.	An array is a homogeneous collection of items of any type such as int, char, etc.
Pointers are allocated at run time.	Arrays are allocated at runtime.
The pointer is a single variable.	An array is a collection of variables of the same type.
Dynamic in Nature	Static in Nature.

## 6. Why do we need to specify the type in the pointer declaration?

Type specification in pointer declaration helps the compiler in dereferencing and pointer arithmetic operations.

### Quizzes:

**SALE!**  
**GeeksforGeeks Courses Upto 25% Off Enroll Now!**



Save 25% on Courses   DSA   Data Structures   Algorithms   Interview Preparation   Data Science   T

# Opaque Pointer in C++

Difficulty Level : Medium • Last Updated : 20 Jan, 2023

[Read](#)   [Discuss](#)   [Courses](#)   [Practice](#)   [Video](#)

Opaque as the name suggests is something we can't see through. e.g. wood is opaque. An opaque pointer is a pointer that points to a data structure whose contents are not exposed at the time of its definition.

The following pointer is opaque. One can't know the data contained in STest structure by looking at the definition.

```
struct STest* pSTest;
```

It is safe to assign NULL to an opaque pointer.

```
pSTest = NULL;
```

## Why Opaque pointer?

There are places where we just want to hint to the compiler that "Hey! This is some data structure that will be used by our clients. Don't worry, clients will provide its implementation while preparing the compilation unit". Such a type of design is robust when we deal with shared code. Please see the below example:

AD

Let's say we are working on an app to deal with images. Since we are living in a world where everything is moving to the cloud and devices are very affordable to buy, we want to develop apps for windows, android, and apple platforms. So, it would be nice to have a good design that is robust, scalable, and flexible as per our requirements. We can have shared code that would be used by all platforms and then different end-point can have platform-specific code. To deal with images, we have a `CImage` class exposing APIs to deal with various image operations (scale, rotate, move, save, etc).

Since all the platforms will be providing the same operations, we would define this class in a header file. But the way an image is handled might differ across platforms. Like Apple can have a different mechanism to access pixels of an image than Windows does. This means that APIs might demand different sets of info to perform operations. So to work on shared code, this is what we would like to do:

**Image.h:** A header file to store class declaration. We will create a header file that will contain a class `CImage` which will provide an API to handle the image operations.

```
// This class provides API to deal with various
// image operations. Different platforms can
// implement these operations in different ways.
class CImage
{
public:
    CImage();
    ~CImage();

    // Opaque pointer

    struct SImageInfo* pImageInfo;

    void Rotate(double angle);

    void Scale(double scaleFactorX,
              double scaleFactorY);

    void Move(int toX, int toY);

private:
```

```
void InitImageInfo();
};
```

**Image.cpp:** Code that will be shared across different endpoints. This file is used to define the constructor and destructor of CImage class. The constructor will call the InitImageInfo() method and the code inside this method will be written in accordance with the Operating System on which it will work on.

```
// Constructor for CImage
```

```
CImage::CImage() {
    InitImageInfo();
}
```

```
// destructor for CImage class
```

```
CImage::~CImage()
{
    // Destroy stuffs here
}
```

**Image\_windows.cpp :** Code specific to Windows operating System will reside in this file.

```
struct SImageInfo {
    // Windows specific DataSet
};

void CImage::InitImageInfo()
{
    pImageInfo = new SImageInfo;
    // Initialize windows specific info here
}

void CImage::Rotate()
{
    // Make use of windows specific SImageInfo
}
```

**Image\_apple.cpp :** Code specific to Mac Operating System will reside in this file.

```

struct SImageInfo {
    // Apple specific DataSet
};

void CImage::InitImageInfo()
{
    pImageInfo = new SImageInfo;

    // Initialize apple specific info here
}

void CImage::Rotate()
{
    // Make use of apple specific SImageInfo
}

```

As it can be seen from the above example **while defining the blueprint of the CImage class we are only mentioning that there is a SImageInfo data structure.**

**The content of SImageInfo is unknown. Now it is the responsibility of clients(windows, apple, android) to define that data structure and use it as per their requirements.** If in the future we want to develop an app for a new end-point 'X', the design is already there. We only need to define SImageInfo for end-point 'X' and use it accordingly.

Please note that the above-explained example is one way of doing this. Design is all about discussion and requirements. A good design is decided to take many factors into account. We can also have platform-specific classes like CImageWindows, and CImageApple and put all platform-specific code there.

### **Downsides of Opaque pointers**

It's also important to note that opaque pointers can have some downsides, as well. For example, because the client code cannot access the implementation details of the object, it may be more difficult to debug or troubleshoot issues that arise. Additionally, opaque pointers can make it more difficult to understand the relationships between objects and their dependencies, which can make it harder to maintain and evolve the codebase over time.

Another potential downside is that opaque pointers can increase the complexity of the codebase, and make it harder to understand how the library is implemented. If a library is large, it can be difficult to understand the relationships between different parts of the codebase, and how they interact.

To mitigate these downsides, it's important to use opaque pointers judiciously, and to provide clear and comprehensive documentation of the library's interface. It's also

important to consider other options, such as PIMPL or Handle-Body idiom, and choose the one that best fits the needs of the project.

In summary, opaque pointers are a technique that can be used to hide the implementation details of an object and provide a level of abstraction in C++. They are useful for hiding the implementation details of an object from the client code, and also for providing a level of abstraction. However, it's important to use them judiciously and to consider other options as well.

It is also important to note that opaque pointers can cause a performance overhead, as it requires extra memory to store the pointer, and an additional level of indirection when accessing the object. It also increases the complexity of memory management as the client code cannot directly deallocate the memory of the object. The library must provide functions to handle the memory management, such as creating and destroying the object, or allocating and deallocating memory for the object.

Additionally, opaque pointers can make it more difficult to write unit tests for the client code as the actual implementation of the object is hidden from the client code. This can make it more difficult to test the client code's interaction with the object.

To mitigate these downsides, it's important to use opaque pointers judiciously, and to provide clear and comprehensive documentation of the library's interface and memory management. It's also important to consider other options, such as PIMPL or Handle-Body idiom, and choose the one that best fits the needs of the project.

In conclusion, opaque pointers are a powerful technique that can be used to hide the implementation details of an object and provide a level of abstraction in C++. However, it's important to use them judiciously and to be aware of the potential downsides, such as performance overhead, memory management, and testing. It's also important to consider other options and choose the one that best fits the needs of the project.

Questions? Keep them coming. We would love to answer.

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

70

## Related Articles

1. Difference between passing pointer to pointer and address of pointer to any function
2. C++ Pointer To Pointer (Double Pointer)

**SALE!**  
**GeeksforGeeks Courses Upto 25% Off Enroll Now!**



Save 25% on Courses DSA Data Structures Algorithms Interview Preparation Data Science T

# References in C++

Difficulty Level : Medium • Last Updated : 30 Mar, 2023

[Read](#) [Discuss\(170+\)](#) [Courses](#) [Practice](#) [Video](#)

When a variable is declared as a reference, it becomes an alternative name for an existing variable. A variable can be declared as a reference by putting '&' in the declaration.

Also, we can define a reference variable as a type of variable that can act as a reference to another variable. '&' is used for signifying the address of a variable or any memory.

Variables associated with reference variables can be accessed either by its name or by the reference variable associated with it.

## Prerequisite: [Pointers in C++](#)

### Syntax:

AD

```
data_type &ref = variable;
```

### Example:

### C++

```
// C++ Program to demonstrate
// use of references
#include <iostream>
```

```

using namespace std;

int main()
{
    int x = 10;

    // ref is a reference to x.
    int& ref = x;

    // Value of x is now changed to 20
    ref = 20;
    cout << "x = " << x << '\n';

    // Value of x is now changed to 30
    x = 30;
    cout << "ref = " << ref << '\n';

    return 0;
}

```

## Output:

```

x = 20
ref = 30

```

## Applications of Reference in C++

There are multiple applications for references in C++, a few of them are mentioned below:

1. Modify the passed parameters in a function
2. Avoiding a copy of large structures
3. In For Each Loop to modify all objects
4. For Each Loop to avoid the copy of objects

### 1. Modify the passed parameters in a function:

If a function receives a reference to a variable, it can modify the value of the variable. For example, the following program variables are swapped using references.

#### Example:

---

## C++

```

// C++ Program to demonstrate
// Passing of references as parameters
#include <iostream>
using namespace std;

```

```
// Function having parameters as
// references
void swap(int& first, int& second)
{
    int temp = first;
    first = second;
    second = temp;
}

// Driver function
int main()
{
    // Variables declared
    int a = 2, b = 3;

    // function called
    swap(a, b);

    // changes can be seen
    // printing both variables
    cout << a << " " << b;
    return 0;
}
```

## Output

3 2

## 2. Avoiding a copy of large structures:

Imagine a function that has to receive a large object. If we pass it without reference, a new copy of it is created which causes a waste of CPU time and memory. We can use references to avoid this.

### Example:

```
struct Student {
    string name;
    string address;
    int rollNo;
}

// If we remove & in below function, a new
// copy of the student object is created.
// We use const to avoid accidental updates
// in the function as the purpose of the function
// is to print s only.
```

```

void print(const Student &s)
{
    cout << s.name << " " << s.address << " " << s.rollNo
        << '\n';
}

```

### 3. In For Each Loop to modify all objects:

We can use references for each loop to modify all elements.

**Example:**

---

**C++**

```

// C++ Program for changing the
// values of elements while traversing
// using references
#include <iostream>
#include <vector>

using namespace std;

// Driver code
int main()
{
    vector<int> vect{ 10, 20, 30, 40 };

    // We can modify elements if we
    // use reference
    for (int& x : vect) {
        x = x + 5;
    }

    // Printing elements
    for (int x : vect) {
        cout << x << " ";
    }
    cout << '\n';

    return 0;
}

```

**Output**

15 25 35 45

### 4. For Each Loop to avoid the copy of objects:

We can use references in each loop to avoid a copy of individual objects when objects are large.

### Example:

---

## C++

```
// C++ Program to use references
// For Each Loop to avoid the
// copy of objects
#include <iostream>
#include <vector>

using namespace std;

// Driver code
int main()
{
    // Declaring vector
    vector<string> vect{ "geeksforgeeks practice",
                         "geeksforgeeks write",
                         "geeksforgeeks ide" };

    // We avoid copy of the whole string
    // object by using reference.
    for (const auto& x : vect) {
        cout << x << '\n';
    }

    return 0;
}
```

## Output

```
geeksforgeeks practice
geeksforgeeks write
geeksforgeeks ide
```

## References vs Pointers

Both references and pointers can be used to change the local variables of one function inside another function. Both of them can also be used to save copying of big objects when passed as arguments to functions or returned from functions, to get efficiency gain. Despite the above similarities, there are the following differences between references and pointers.

1. A pointer can be declared as void but a reference can never be void For example

```
int a = 10;
void* aa = &a; // it is valid
void& ar = a; // it is not valid
```

**2.** The pointer variable has n-levels/multiple levels of indirection i.e. single-pointer, double-pointer, triple-pointer. Whereas, the reference variable has only one/single level of indirection. The following code reveals the mentioned points:

- 3.** Reference variables cannot be updated.
- 4.** Reference variable is an internal pointer.
- 5.** Declaration of a Reference variable is preceded with the '&' symbol ( but do not read it as "address of").

### Example:

---

## C++

```
// C++ Program to demonstrate
// references and pointers
#include <iostream>
using namespace std;

// Driver Code
int main()
{
    // simple or ordinary variable.
    int i = 10;

    // single pointer
    int* p = &i;

    // double pointer
    int** pt = &p;

    // triple pointer
    int*** ptr = &pt;

    // All the above pointers differ in the value they store
    // or point to.
    cout << "i = " << i << "\t"
        << "p = " << p << "\t"
        << "pt = " << pt << "\t"
        << "ptr = " << ptr << '\n';

    // simple or ordinary variable
    int a = 5;
    int& S = a;
    int& S0 = S;
    int& S1 = S0;
```

```

// All the references do not differ in their
// values as they all refer to the same variable.
cout << "a = " << a << "\t"
    << "S = " << S << "\t"
    << "S0 = " << S0 << "\t"
    << "S1 = " << S1 << '\n';

return 0;
}

```

## Output

```
i = 10      p = 0x7ffecfe7c07c      pt = 0x7ffecfe7c080      ptr = 0x7ffecfe7c088
a = 5      S = 5      S0 = 5      S1 = 5
```

## Limitations of References

- Once a reference is created, it cannot be later made to reference another object; it cannot be reset. This is often done with pointers.
- References cannot be NULL. Pointers are often made NULL to indicate that they are not pointing to any valid thing.
- A reference must be initialized when declared. There is no such restriction with pointers.

Due to the above limitations, references in C++ cannot be used for implementing data structures like Linked List, Tree, etc. In Java, references don't have the above restrictions and can be used to implement all data structures. References being more powerful in Java is the main reason Java doesn't need pointers.

## Advantages of using References

- Safer:** Since references must be initialized, wild references like wild pointers are unlikely to exist. It is still possible to have references that don't refer to a valid location (See questions 5 and 6 in the below exercise)
- Easier to use:** References don't need a dereferencing operator to access the value. They can be used like normal variables. The '&' operator is needed only at the time of declaration. Also, members of an object reference can be accessed with the dot operator ('.'), unlike pointers where the arrow operator (->) is needed to access members.

Together with the above reasons, there are a few places like the copy constructor argument where a pointer cannot be used. Reference must be used to pass the argument in the copy constructor. Similarly, references must be used for overloading some operators like ++.

## Exercise with Answers

### Question 1 :

---

#### C++

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

int& fun()
{
    static int x = 10;
    return x;
}

int main()
{
    fun() = 30;
    cout << fun();
    return 0;
}
```

#### Output

30

### Question 2

---

#### C++

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

int fun(int& x) { return x; }

int main()
{
    cout << fun(10);
    return 0;
}
```

#### Output:

```
./3337ee98-ae6e-4792-8128-7c879288221f.cpp: In function 'int main()':
./3337ee98-ae6e-4792-8128-7c879288221f.cpp:8:19: error: invalid
initialization of non-const reference of type 'int&' from an rvalue of type
'int'
```

```

cout << fun(10);
^

./3337ee98-ae6e-4792-8128-7c879288221f.cpp:4:5: note: in passing argument 1
of 'int fun(int&)'
int fun(int& x) { return x; }

```

### Question 3

---

#### C++

```

#include <iostream>
using namespace std;

void swap(char*& str1, char*& str2)
{
    char* temp = str1;
    str1 = str2;
    str2 = temp;
}

int main()
{
    char* str1 = "GEEKS";
    char* str2 = "FOR GEEKS";
    swap(str1, str2);
    cout << "str1 is " << str1 << '\n';
    cout << "str2 is " << str2 << '\n';
    return 0;
}

```

#### Output

str1 is FOR GEEKS  
str2 is GEEKS

### Question 4

---

#### C++

```

#include <iostream>
using namespace std;

int main()
{
    int x = 10;
    int* ptr = &x;
    int&* ptr1 = ptr;
}

```

**Output:**

```
./18074365-ebdc-4b13-81f2-cfc42bb4b035.cpp: In function 'int main()':
./18074365-ebdc-4b13-81f2-cfc42bb4b035.cpp:8:11: error: cannot declare
pointer to 'int&'
    int&* ptr1 = ptr;
```

**Question 5****C++**

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

int main()
{
    int* ptr = NULL;
    int& ref = *ptr;
    cout << ref << '\n';
}
```

**Output:**

```
timeout: the monitored command dumped core
/bin/bash: line 1:    34 Segmentation fault      timeout 15s ./372da97e-
346c-4594-990f-14edda1f5021 < 372da97e-346c-4594-990f-14edda1f5021.in
```

**Question 6****C++**

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

int& fun()
{
    int x = 10;
    return x;
}

int main()
{
    fun() = 30;
    cout << fun();
    return 0;
}
```

## Output

0

## Related Articles

- [Pointers vs References in C++](#)
- [When do we pass arguments by reference or pointer?](#)
- [Can references refer to an invalid location in C++?](#)
- [Passing by pointer Vs Passing by Reference in C++](#)

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

291

## Related Articles

1. [lvalues references and rvalues references in C++ with Examples](#)
2. [Can References Refer to Invalid Location in C++?](#)
3. [Default Assignment Operator and References in C++](#)
4. [Pointers and References in C++](#)
5. [Overloads of the Different References in C++](#)
6. [Pointers vs References in C++](#)
7. [C++ Error - Does not name a type](#)
8. [Execution Policy of STL Algorithms in Modern C++](#)
9. [C++ Program To Print Pyramid Patterns](#)
10. [Jagged Arrays in C++](#)

Previous

Next

**SALE!**  
**GeeksforGeeks Courses Upto 25% Off Enroll Now!**

[Save 25% on Courses](#)[DSA](#)[Data Structures](#)[Algorithms](#)[Interview Preparation](#)[Data Science](#)[T](#)

# 'this' pointer in C++

Difficulty Level : Medium • Last Updated : 09 Aug, 2019

[Read](#)[Discuss\(130+\)](#)[Courses](#)[Practice](#)[Video](#)

To understand 'this' pointer, it is important to know how objects look at functions and data members of a class.

1. Each object gets its own copy of the data member.
2. All-access the same function definition as present in the code segment.

Meaning each object gets its own copy of data members and all objects share a single copy of member functions.

Then now question is that if only one copy of each member function exists and is used by multiple objects, how are the proper data members are accessed and updated?

The compiler supplies an implicit pointer along with the names of the functions as 'this'.

The 'this' pointer is passed as a hidden argument to all nonstatic member function calls and is available as a local variable within the body of all nonstatic functions. 'this' pointer is not available in static member functions as static member functions can be called without any object (with class name).

For a class X, the type of this pointer is 'X\*'. Also, if a member function of X is declared as const, then the type of this pointer is 'const X \*' (see [this GFact](#))

In the early version of C++ would let 'this' pointer to be changed; by doing so a programmer could change which object a method was working on. This feature was eventually removed, and now this in C++ is an r-value.

C++ lets object destroy themselves by calling the following code :

```
delete this;
```

As Stroustrup said 'this' could be the reference than the pointer, but the reference was not present in the early version of C++. If 'this' is implemented as a reference then, the above problem could be avoided and it could be safer than the pointer.

AD

Following are the situations where 'this' pointer is used:

### 1) When local variable's name is same as member's name

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

/* local variable is same as a member's name */
class Test
{
private:
    int x;
public:
    void setX (int x)
    {
        // The 'this' pointer is used to retrieve the object's x
        // hidden by the local variable 'x'
        this->x = x;
    }
    void print() { cout << "x = " << x << endl; }
};

int main()
{
    Test obj;
    int x = 20;
    obj.setX(x);
    obj.print();
    return 0;
}
```

Output:

```
x = 20
```

For constructors, [initializer list](#) can also be used when parameter name is same as member's name.

## 2) To return reference to the calling object

```
/* Reference to the calling object can be returned */
Test& Test::func ()
{
    // Some processing
    return *this;
}
```

When a reference to a local object is returned, the returned reference can be used to **chain function calls** on a single object.

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

class Test
{
private:
    int x;
    int y;
public:
    Test(int x = 0, int y = 0) { this->x = x; this->y = y; }
    Test &setX(int a) { x = a; return *this; }
    Test &setY(int b) { y = b; return *this; }
    void print() { cout << "x = " << x << " y = " << y << endl; }
};

int main()
{
    Test obj1(5, 5);

    // Chained function calls. All calls modify the same object
    // as the same object is returned by reference
    obj1.setX(10).setY(20);

    obj1.print();
    return 0;
}
```

Output:

x = 10 y = 20

### Exercise:

Predict the output of following programs. If there are compilation errors, then fix them.

## Question 1

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

class Test
{
private:
    int x;
public:
    Test(int x = 0) { this->x = x; }
    void change(Test *t) { this = t; }
    void print() { cout << "x = " << x << endl; }
};

int main()
{
    Test obj(5);
    Test *ptr = new Test (10);
    obj.change(ptr);
    obj.print();
    return 0;
}
```

## Question 2

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

class Test
{
private:
    int x;
    int y;
public:
    Test(int x = 0, int y = 0) { this->x = x; this->y = y; }
    static void fun1() { cout << "Inside fun1()"; }
    static void fun2() { cout << "Inside fun2()"; this->fun1(); }
};

int main()
{
    Test obj;
    obj.fun2();
    return 0;
}
```

### Question 3

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

class Test
{
private:
    int x;
    int y;
public:
    Test (int x = 0, int y = 0) { this->x = x; this->y = y; }
    Test setX(int a) { x = a; return *this; }
    Test setY(int b) { y = b; return *this; }
    void print() { cout << "x = " << x << " y = " << y << endl; }
};

int main()
{
    Test obj1;
    obj1.setX(10).setY(20);
    obj1.print();
    return 0;
}
```

### Question 4

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

class Test
{
private:
    int x;
    int y;
public:
    Test(int x = 0, int y = 0) { this->x = x; this->y = y; }
    void setX(int a) { x = a; }
    void setY(int b) { y = b; }
    void destroy() { delete this; }
    void print() { cout << "x = " << x << " y = " << y << endl; }
};

int main()
{
    Test obj;
    obj.destroy();
    obj.print();
    return 0;
}
```

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

248

## Related Articles

1. [C – Pointer to Pointer \(Double Pointer\)](#)

---
2. [Difference between passing pointer to pointer and address of pointer to any function](#)

---
3. [C++ Pointer To Pointer \(Double Pointer\)](#)

---
4. [Pointer to an Array | Array Pointer](#)

---
5. [What is a Pointer to a Null pointer](#)

---
6. [Difference between Dangling pointer and Void pointer](#)

---
7. [How to declare a pointer to a function?](#)

---
8. [Declare a C/C++ function returning pointer to array of integer pointers](#)

---
9. [Type of 'this' Pointer in C++](#)

---
10. [Multidimensional Pointer Arithmetic in C/C++](#)

[Previous](#)[Next](#)

## Article Contributed By :



GeeksforGeeks

## Vote for difficulty

Current difficulty : [Medium](#)

[Easy](#)[Normal](#)[Medium](#)[Hard](#)[Expert](#)

**SALE!**  
**GeeksforGeeks Courses Upto 25% Off Enroll Now!**



Save 25% on Courses DSA Data Structures Algorithms Interview Preparation Data Science T

# Smart Pointers in C++

Difficulty Level : Medium • Last Updated : 16 Mar, 2023

[Read](#) [Discuss\(30+\)](#) [Courses](#) [Practice](#) [Video](#)

## Prerequisite: [Pointers in C++](#)

Pointers are used for accessing the resources which are external to the program – like heap memory. So, for accessing the heap memory (if anything is created inside heap memory), pointers are used. When accessing any external resource we just use a copy of the resource. If we make any changes to it, we just change it in the copied version. But, if we use a pointer to the resource, we'll be able to change the original resource.

## Problems with Normal Pointers

Some Issues with normal pointers in C++ are as follows:

- **Memory Leaks:** This occurs when memory is repeatedly allocated by a program but never freed. This leads to excessive memory consumption and eventually leads to a system crash.
- **Dangling Pointers:** A [dangling pointer](#) is a pointer that occurs at the time when the object is de-allocated from memory without modifying the value of the pointer.
- **Wild Pointers:** Wild pointers are pointers that are declared and allocated memory but the pointer is never initialized to point to any valid object or address.
- **Data Inconsistency:** Data inconsistency occurs when some data is stored in memory but is not updated in a consistent manner.
- **Buffer Overflow:** When a pointer is used to write data to a memory address that is outside of the allocated memory block. This leads to the corruption of data which can be exploited by malicious attackers.

## Example:

AD

## C++

```
// C++ program to demonstrate working of a Pointers
#include <iostream>
using namespace std;

class Rectangle {
private:
    int length;
    int breadth;
};

void fun()
{
    // By taking a pointer p and
    // dynamically creating object
    // of class rectangle
    Rectangle* p = new Rectangle();
}

int main()
{
    // Infinite Loop
    while (1) {
        fun();
    }
}
```

### Output:

Memory limit exceeded

**Explanation:** In function *fun*, it creates a pointer that is pointing to the *Rectangle* object. The object *Rectangle* contains two integers, *length*, and *breadth*. When the function *fun* ends, *p* will be destroyed as it is a local variable. But, the memory it consumed won't be deallocated because we forgot to use *delete p*; at the end of the function. That means the memory won't be free to be used by other resources. But, we don't need the variable anymore, we need the memory.

In function *main*, *fun* is called in an infinite loop. That means it'll keep creating *p*. It'll allocate more and more memory but won't free them as we didn't deallocate it. The memory that's wasted can't be used again. Which is a memory leak. The entire *heap* memory may become useless for this reason.

## Smart Pointers

As we've known unconsciously not deallocating a pointer causes a memory leak that may lead to a crash of the program. Languages Java, C# has *Garbage Collection Mechanisms* to smartly deallocate unused memory to be used again. The programmer doesn't have to worry about any memory leaks. C++ comes up with its own mechanism that's *Smart Pointer*. When the object is destroyed it frees the memory as well. So, we don't need to delete it as Smart Pointer does will handle it.

A *Smart Pointer* is a wrapper class over a pointer with an operator like **\*** and **->** overloaded. The objects of the smart pointer class look like normal pointers. But, unlike *Normal Pointers* it can deallocate and free destroyed object memory.

The idea is to take a class with a pointer, destructor, and overloaded operators like **\*** and **->**. Since the destructor is automatically called when an object goes out of scope, the dynamically allocated memory would automatically be deleted (or the reference count can be decremented).

### **Example:**

---

## C++

```
// C++ program to demonstrate the working of Smart Pointer
#include <iostream>
using namespace std;

class SmartPtr {
    int* ptr; // Actual pointer
public:
    // Constructor: Refer https://www.geeksforgeeks.org/g-fact-93/
    // for use of explicit keyword
    explicit SmartPtr(int* p = NULL) { ptr = p; }

    // Destructor
    ~SmartPtr() { delete (ptr); }

    // Overloading dereferencing operator
    int& operator*() { return *ptr; }
};

int main()
{
    SmartPtr ptr(new int());
    *ptr = 20;
```

```

cout << *ptr;

// We don't need to call delete ptr: when the object
// ptr goes out of scope, the destructor for it is automatically
// called and destructor does delete ptr.

return 0;
}

```

## Output

20

## Difference Between Pointers and Smart Pointers

Pointer	Smart Pointer
A pointer is a variable that maintains a memory address as well as data type information about that memory location. A pointer is a variable that points to something in memory.	It's a pointer-wrapping stack-allocated object. Smart pointers, in plain terms, are classes that wrap a pointer, or scoped pointers.
It is not destroyed in any form when it goes out of its scope	It destroys itself when it goes out of its scope
Pointers are not so efficient as they don't support any other feature.	Smart pointers are more efficient as they have an additional feature of memory management.
They are very labor-centric/manual.	They are automatic/pre-programmed in nature.

**Note:** This only works for int. So, we'll have to create Smart Pointer for every object?  
**No,** there's a solution, [Template](#). In the code below as you can see T can be of any type.

## Example:

## C++

```
// C++ program to demonstrate the working of Template and
// overcome the issues which we are having with pointers
#include <iostream>
using namespace std;

// A generic smart pointer class
template <class T> class SmartPtr {
    T* ptr; // Actual pointer
public:
    // Constructor
    explicit SmartPtr(T* p = NULL) { ptr = p; }

    // Destructor
    ~SmartPtr() { delete (ptr); }

    // Overloading dereferencing operator
    T& operator*() { return *ptr; }

    // Overloading arrow operator so that
    // members of T can be accessed
    // like a pointer (useful if T represents
    // a class or struct or union type)
    T* operator->() { return ptr; }
};

int main()
{
    SmartPtr<int> ptr(new int());
    *ptr = 20;
    cout << *ptr;
    return 0;
}
```

### Output:

20

**Note:** Smart pointers are also useful in the management of resources, such as file handles or network sockets.

## Types of Smart Pointers

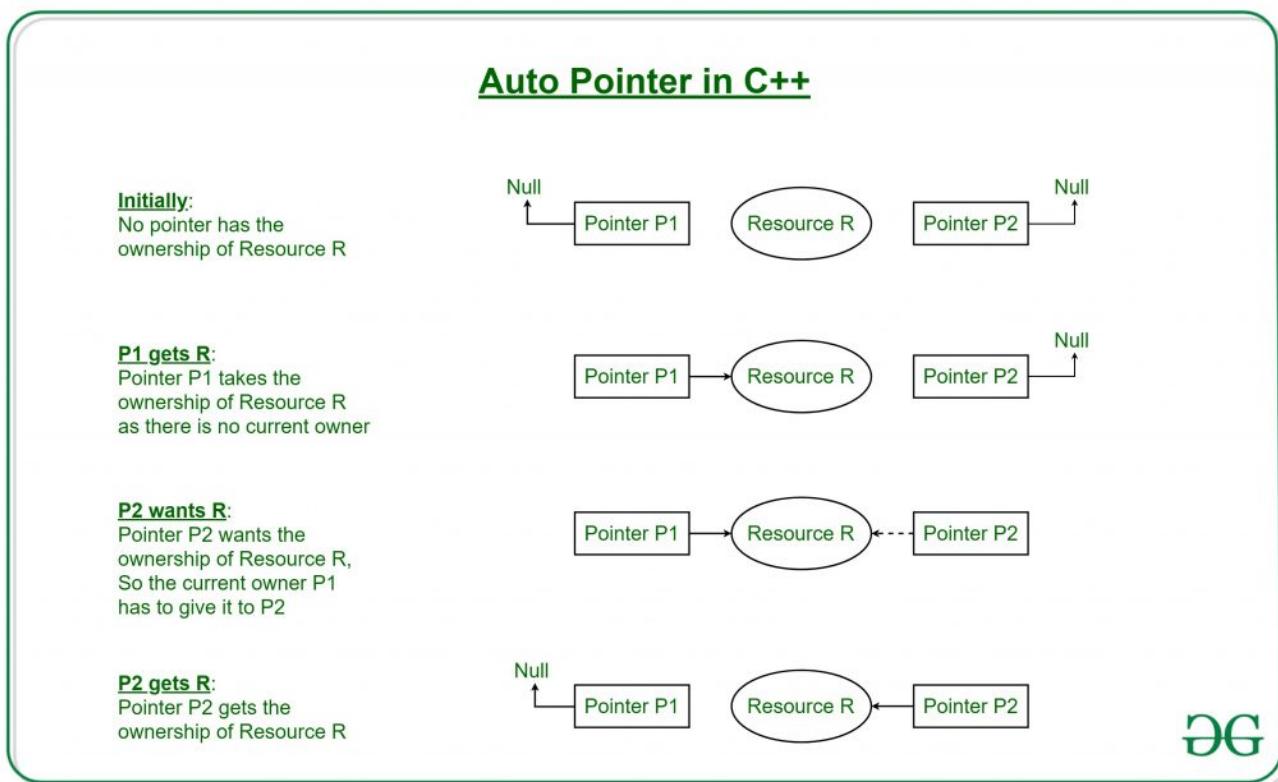
C++ libraries provide implementations of smart pointers in the following types:

- auto\_ptr
- unique\_ptr
- shared\_ptr
- weak\_ptr

## auto\_ptr

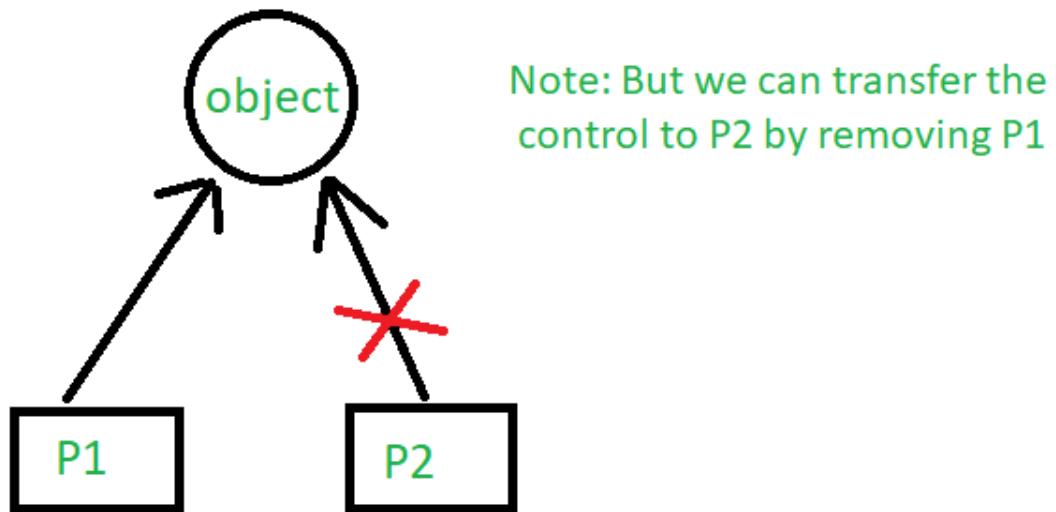
Using auto\_ptr, you can manage objects obtained from new expressions and delete them when auto\_ptr itself is destroyed. When an object is described through auto\_ptr it stores a pointer to a single allocated object.

**Note:** This class template is deprecated as of C++11. unique\_ptr is a new facility with a similar functionality, but with improved security.



## unique\_ptr

unique\_ptr stores one pointer only. We can assign a different object by removing the current object from the pointer.

**Example:****C++**

```
// C++ program to demonstrate the working of unique_ptr
// Here we are showing the unique_pointer is pointing to P1.
// But, then we remove P1 and assign P2 so the pointer now
// points to P2.

#include <iostream>
using namespace std;
// Dynamic Memory management library
#include <memory>

class Rectangle {
    int length;
    int breadth;

public:
    Rectangle(int l, int b)
    {
        length = l;
        breadth = b;
    }

    int area() { return length * breadth; }
};

int main()
{
// --\ Smart Pointer
    unique_ptr<Rectangle> P1(new Rectangle(10, 5));
    cout << P1->area() << endl; // This'll print 50

    // unique_ptr<Rectangle> P2(P1);
}
```

```

unique_ptr<Rectangle> P2;
P2 = move(P1);

// This'll print 50
cout << P2->area() << endl;

// cout<<P1->area()<<endl;
return 0;
}

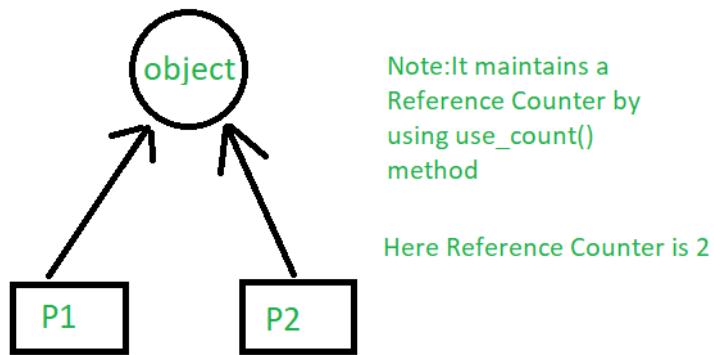
```

## Output

50  
50

## shared\_ptr

By using *shared\_ptr* more than one pointer can point to this one object at a time and it'll maintain a **Reference Counter** using the ***use\_count()* method**.




---

## C++

```

// C++ program to demonstrate the working of shared_ptr
// Here both smart pointer P1 and P2 are pointing to the
// object Addition to which they both maintain a reference
// of the object
#include <iostream>
using namespace std;
// Dynamic Memory management library
#include <memory>

class Rectangle {
    int length;

```

```

int breadth;

public:
    Rectangle(int l, int b)
    {
        length = l;
        breadth = b;
    }

    int area() { return length * breadth; }

int main()
{
    //---\ Smart Pointer
    shared_ptr<Rectangle> P1(new Rectangle(10, 5));
    // This'll print 50
    cout << P1->area() << endl;

    shared_ptr<Rectangle> P2;
    P2 = P1;

    // This'll print 50
    cout << P2->area() << endl;

    // This'll now not give an error,
    cout << P1->area() << endl;

    // This'll also print 50 now
    // This'll print 2 as Reference Counter is 2
    cout << P1.use_count() << endl;
    return 0;
}

```

## Output

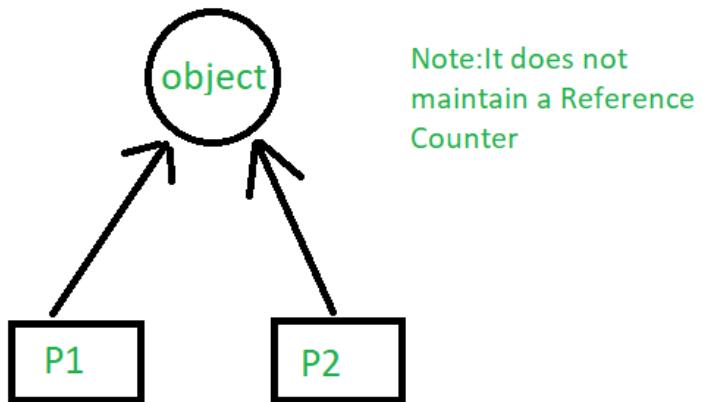
```

50
50
50
2

```

## **weak\_ptr**

Weak\_ptr is a smart pointer that holds a non-owning reference to an object. It's much more similar to shared\_ptr except it'll not maintain a **Reference Counter**. In this case, a pointer will not have a stronghold on the object. The reason is if suppose pointers are holding the object and requesting for other objects then they may form a **Deadlock**.



## C++

```
// C++ program to demonstrate the working of weak_ptr
// Here both smart pointer P1 and P2 are pointing to the
// object Addition to which they both does not maintain
// a reference of the object
#include <iostream>
using namespace std;
// Dynamic Memory management library
#include <memory>

class Rectangle {
    int length;
    int breadth;

public:
    Rectangle(int l, int b)
    {
        length = l;
        breadth = b;
    }

    int area() { return length * breadth; }
};

int main()
{
    //---\| Smart Pointer
    shared_ptr<Rectangle> P1(new Rectangle(10, 5));

    // This'll print 50
    cout << P1->area() << endl;

    auto P2 = P1;
}
```

```
// P2 = P1;

// This'll print 50
cout << P2->area() << endl;

// This'll now not give an error,
cout << P1->area() << endl;

// This'll also print 50 now
// This'll print 2 as Reference Counter is 2
cout << P1.use_count() << endl;
return 0;
}
```

## Output

50  
50  
50  
2

C++ libraries provide implementations of smart pointers in the form of [auto\\_ptr](#), [unique\\_ptr](#), [shared\\_ptr](#), and [weak\\_ptr](#)

152

## Related Articles

1. Difference between constant pointer, pointers to constant, and constant pointers to constants

---

2. Trie Data Structure using smart pointer and OOP in C++

---

3. What are Wild Pointers? How can we avoid?

---

4. Declare a C/C++ function returning pointer to array of integer pointers

---

5. C++ Pointers

---

6. Computing Index Using Pointers Returned By STL Functions in C++

---

7. Applications of Pointers in C/C++

---

8. C++ Program to compare two string using pointers

**SALE!**  
**GeeksforGeeks Courses Upto 25% Off Enroll Now!**



Save 25% on Courses   DSA   Data Structures   Algorithms   Interview Preparation   Data Science   T

# Pointers vs References in C++

Difficulty Level : Easy • Last Updated : 11 Mar, 2023

[Read](#)   [Discuss](#)   [Courses](#)   [Practice](#)   [Video](#)

Prerequisite: [Pointers, References](#)

C and C++ support pointers, which is different from most other programming languages such as Java, Python, Ruby, Perl and PHP as they only support references. But interestingly, C++, along with pointers, also supports references.

On the surface, both references and pointers are very similar as both are used to have one variable provide access to another. With both providing lots of the same capabilities, it's often unclear what is different between these mechanisms. In this article, I will try to illustrate the differences between pointers and references.

**Pointers:** A pointer is a variable that holds the memory address of another variable. A pointer needs to be dereferenced with the `*` operator to access the memory location it points to.

AD

**References:** A reference variable is an alias, that is, another name for an already existing variable. A reference, like a pointer, is also implemented by storing the address of an object.

A reference can be thought of as a constant pointer (not to be confused with a pointer to a

constant value!) with automatic indirection, i.e., the compiler will apply the `*` operator for you.

```
int i = 3;

// A pointer to variable i or "stores the address of i"
int *ptr = &i;

// A reference (or alias) for i.
int &ref = i;
```

## Differences:

1. **Initialization:** A pointer can be initialized in this way:

```
int a = 10;
int *p = &a;
// OR
int *p;
p = &a;
```

We can declare and initialize pointer at same step or in multiple line.

2. While in references,

```
int a = 10;
int &p = a; // It is correct
// but
int &p;
p = a; // It is incorrect as we should declare and initialize references at
single step
```

**NOTE:** This difference may vary from compiler to compiler. The above difference is with respect to Turbo IDE.

3. **Reassignment:** A pointer can be re-assigned. This property is useful for the implementation of data structures like a linked list, a tree, etc. See the following example:

```
int a = 5;
int b = 6;
int *p;
p = &a;
p = &b;
```

4. On the other hand, a reference cannot be re-assigned, and must be assigned at initialization.

```
int a = 5;
int b = 6;
int &p = a;
int &p = b; // This will throw an error of "multiple declaration is not allowed"

// However it is valid statement,
int &q = p;
```

5. **Memory Address:** A pointer has its own memory address and size on the stack, whereas a reference shares the same memory address with the original variable but also takes up some space on the stack.

```
int &p = a;
cout << &p << endl << &a;
```

6. **NULL value:** A pointer can be assigned NULL directly, whereas a reference cannot be. The constraints associated with references (no NULL, no reassignment) ensure that the underlying operations do not run into an exception situation.

7. **Indirection:** You can have a pointer to pointer (known as a double pointer) offering extra levels of indirection, whereas references only offer one level of indirection. For example,

```
In Pointers,
int a = 10;
int *p;
int **q; // It is valid.
p = &a;
q = &p;

// Whereas in references,
int &p = a;
int &&q = p; // It is reference to reference, so it is an error
```

8. **Arithmetic operations:** Various arithmetic operations can be performed on pointers, whereas there is no such thing called Reference Arithmetic (however, you can perform pointer arithmetic on the address of an object pointed to by a reference, as in `&obj + 5`).

### **Tabular form of difference between References and Pointers in C++**

	<b>References</b>	<b>Pointers</b>
<b>Reassignment</b>	The variable cannot be reassigned in Reference.	The variable can be reassigned in Pointers.
<b>Memory Address</b>	It shares the same address as the original variable.	Pointers have their own memory address.
<b>Work</b>	It is referring to another variable.	It is storing the address of the variable.
<b>Null Value</b>	It does not have null value.	It can have value assigned as null.
<b>Arguments</b>	This variable is referenced by the method pass by value.	The pointer does it work by the method known as pass by reference.

### When to use What

The performances are exactly the same as references are implemented internally as pointers. But still, you can keep some points in your mind to decide when to use what:

- Use references:
  - In function parameters and return types.
- Use pointers:
  - If pointer arithmetic or passing a NULL pointer is needed. For example, for arrays (Note that accessing an array is implemented using pointer arithmetic).
  - To implement data structures like a linked list, a tree, etc. and their algorithms. This is so because, in order to point to different cells, we have to use the concept of pointers.

Quoted in C++ FAQ Lite: Use references when you can, and pointers when you have to.

References are usually preferred over pointers whenever you don't need "reseating". This usually means that references are most useful in a class's public interface. References typically appear on the skin of an object, and pointers on the inside.

The exception to the above is where a function's parameter or return value needs a "sentinel" reference – a reference that does not refer to an object. This is usually best done by returning/taking a pointer, and giving the "nullptr" value this special significance (references must always alias objects, not a dereferenced null pointer).

## Related Article:

### [When do we pass arguments as Reference or Pointers?](#)

This article is contributed by **Rishav Raj**. If you like GeeksforGeeks and would like to contribute, you can also write an article using [write.geeksforgeeks.org](#) or mail your article to review-team@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

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

217

## Related Articles

1. [lvalues references and rvalues references in C++ with Examples](#)

---
2. [Pointers and References in C++](#)

---
3. [Difference between constant pointer, pointers to constant, and constant pointers to constants](#)

---
4. [Can References Refer to Invalid Location in C++?](#)

---
5. [Default Assignment Operator and References in C++](#)

---
6. [C++ | References | Question 1](#)

---
7. [C++ | References | Question 6](#)

---
8. [C++ | References | Question 6](#)

---
9. [C++ | References | Question 4](#)

---
10. [C++ | References | Question 6](#)

[Previous](#)

[Next](#)

## Article Contributed By :



GeeksforGeeks

**SALE!**  
**GeeksforGeeks Courses Upto 25% Off Enroll Now!**



Save 25% on Courses   DSA   Data Structures   Algorithms   Interview Preparation   Data Science   T

# Object Oriented Programming in C++

Difficulty Level : Easy • Last Updated : 11 Mar, 2023

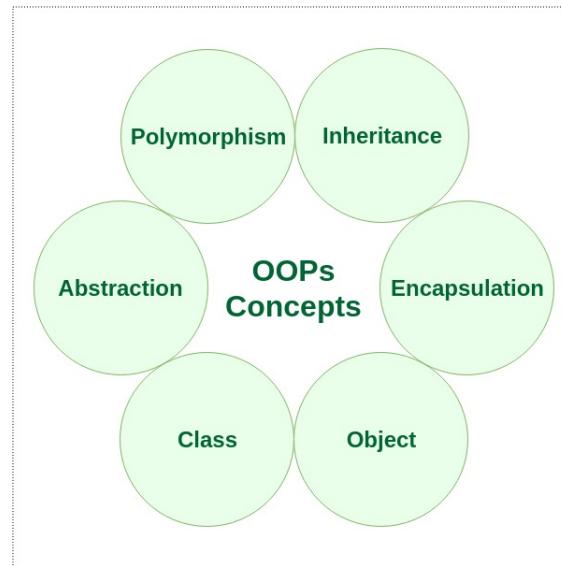
[Read](#)   [Discuss](#)   [Courses](#)   [Practice](#)   [Video](#)

Object-oriented programming – As the name suggests uses objects in programming. Object-oriented programming aims to implement real-world entities like inheritance, hiding, polymorphism, etc. in programming. The main aim of OOP is to bind together the data and the functions that operate on them so that no other part of the code can access this data except that function.

There are some basic concepts that act as the building blocks of OOPs i.e.

1. Class
2. Objects
3. Encapsulation
4. Abstraction
5. Polymorphism
6. Inheritance
7. Dynamic Binding
8. Message Passing

## Characteristics of an Object-Oriented Programming Language



## Class

The building block of C++ that leads to Object-Oriented programming is a Class. It is a user-defined data type, which holds its own data members and member functions, which can be accessed and used by creating an instance of that class. A class is like a blueprint for an object. For Example: Consider the Class of Cars. There may be many cars with different names and brands but all of them will share some common properties like all of them will have 4 wheels, Speed Limit, Mileage range, etc. So here, the Car is the class, and wheels, speed limits, and mileage are their properties.

AD

- A Class is a user-defined data type that has data members and member functions.
- Data members are the data variables and member functions are the functions used to manipulate these variables together these data members and member functions define the properties and behavior of the objects in a Class.
- In the above example of class Car, the data member will be speed limit, mileage, etc and member functions can apply brakes, increase speed, etc.

We can say that a **Class in C++** is a blueprint representing a group of objects which shares some common properties and behaviors.

## Object

An Object is an identifiable entity with some characteristics and behavior. An Object is an instance of a Class. When a class is defined, no memory is allocated but when it is instantiated (i.e. an object is created) memory is allocated.

## C++

```
// C++ Program to show the syntax/working of Objects as a
// part of Object Oriented Programming
#include <iostream>
using namespace std;

class person {
    char name[20];
    int id;

public:
    void getdetails() {}
};

int main()
{
    person p1; // p1 is a object
    return 0;
}
```

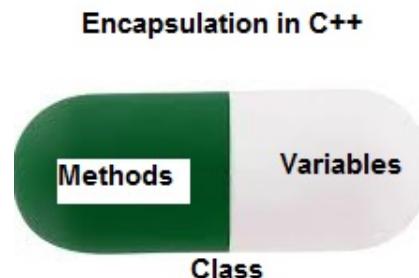
Objects take up space in memory and have an associated address like a record in pascal or structure or union. When a program is executed the objects interact by sending messages to one another. Each object contains data and code to manipulate the data. Objects can interact without having to know details of each other's data or code, it is sufficient to know the type of message accepted and the type of response returned by the objects.

To know more about C++ Objects and Classes, refer to this article – [C++ Classes and Objects](#)

## Encapsulation

In normal terms, Encapsulation is defined as wrapping up data and information under a single unit. In Object-Oriented Programming, Encapsulation is defined as binding together the data and the functions that manipulate them. Consider a real-life example of encapsulation, in a company, there are different sections like the accounts section, finance section, sales section, etc. The finance section handles all the financial transactions and keeps records of all the data related to finance. Similarly, the sales section handles all the sales-related activities and keeps records of all the sales. Now there may arise a situation when for some reason an official from the finance section needs all the data about sales in a particular month. In this case, he is not allowed to directly access the data of the sales

section. He will first have to contact some other officer in the sales section and then request him to give the particular data. This is what encapsulation is. Here the data of the sales section and the employees that can manipulate them are wrapped under a single name "sales section".



*Encapsulation in C++*

Encapsulation also leads to *data abstraction or data hiding*. Using encapsulation also hides the data. In the above example, the data of any of the sections like sales, finance, or accounts are hidden from any other section.

To know more about encapsulation, refer to this article – [Encapsulation in C++](#)

## Abstraction

Data abstraction is one of the most essential and important features of object-oriented programming in C++. Abstraction means displaying only essential information and hiding the details. Data abstraction refers to providing only essential information about the data to the outside world, hiding the background details or implementation. Consider a real-life example of a man driving a car. The man only knows that pressing the accelerator will increase the speed of the car or applying brakes will stop the car but he does not know how on pressing the accelerator the speed is actually increasing, he does not know about the inner mechanism of the car or the implementation of an accelerator, brakes, etc. in the car. This is what abstraction is.

- **Abstraction using Classes:** We can implement Abstraction in C++ using classes. The class helps us to group data members and member functions using available access specifiers. A Class can decide which data member will be visible to the outside world and which is not.
- **Abstraction in Header files:** One more type of abstraction in C++ can be header files. For example, consider the pow() method present in math.h header file. Whenever we need to calculate the power of a number, we simply call the function pow() present in the math.h header file and pass the numbers as arguments without knowing the underlying algorithm according to which the function is actually calculating the power of numbers.

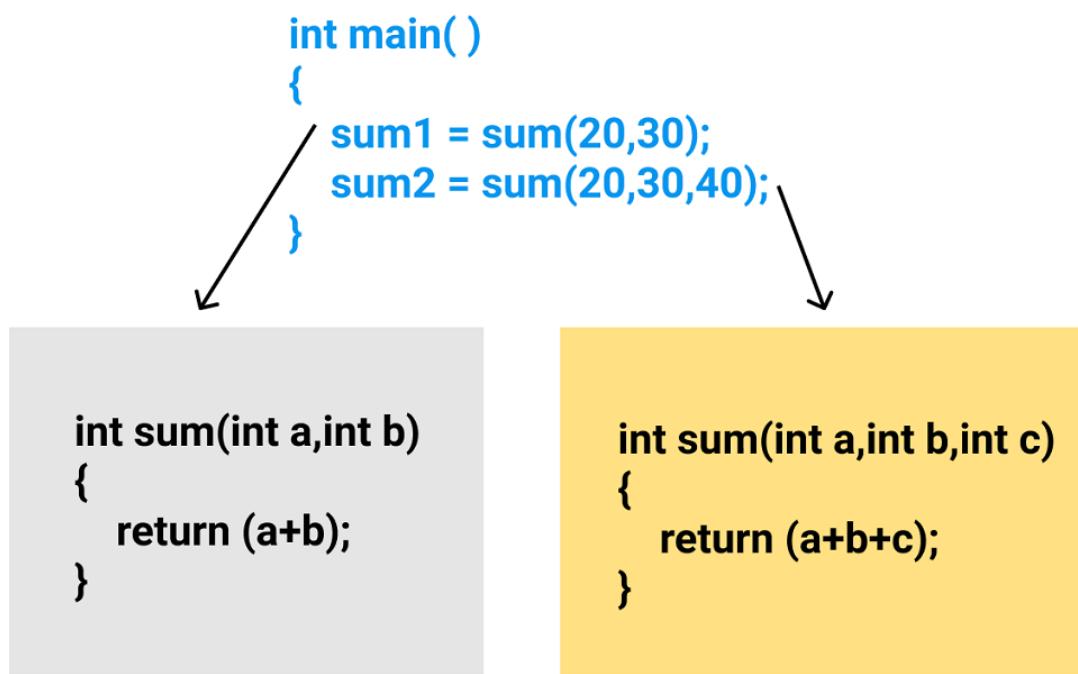
To know more about C++ abstraction, refer to this article – [Abstraction in C++](#)

## Polymorphism

The word polymorphism means having many forms. In simple words, we can define polymorphism as the ability of a message to be displayed in more than one form. A person at the same time can have different characteristics. A man at the same time is a father, a husband, and an employee. So the same person possesses different behavior in different situations. This is called polymorphism. An operation may exhibit different behaviors in different instances. The behavior depends upon the types of data used in the operation. C++ supports operator overloading and function overloading.

- **Operator Overloading:** The process of making an operator exhibit different behaviors in different instances is known as operator overloading.
- **Function Overloading:** Function overloading is using a single function name to perform different types of tasks. Polymorphism is extensively used in implementing inheritance.

**Example:** Suppose we have to write a function to add some integers, sometimes there are 2 integers, and sometimes there are 3 integers. We can write the Addition Method with the same name having different parameters, the concerned method will be called according to parameters.



Polymorphism in C++

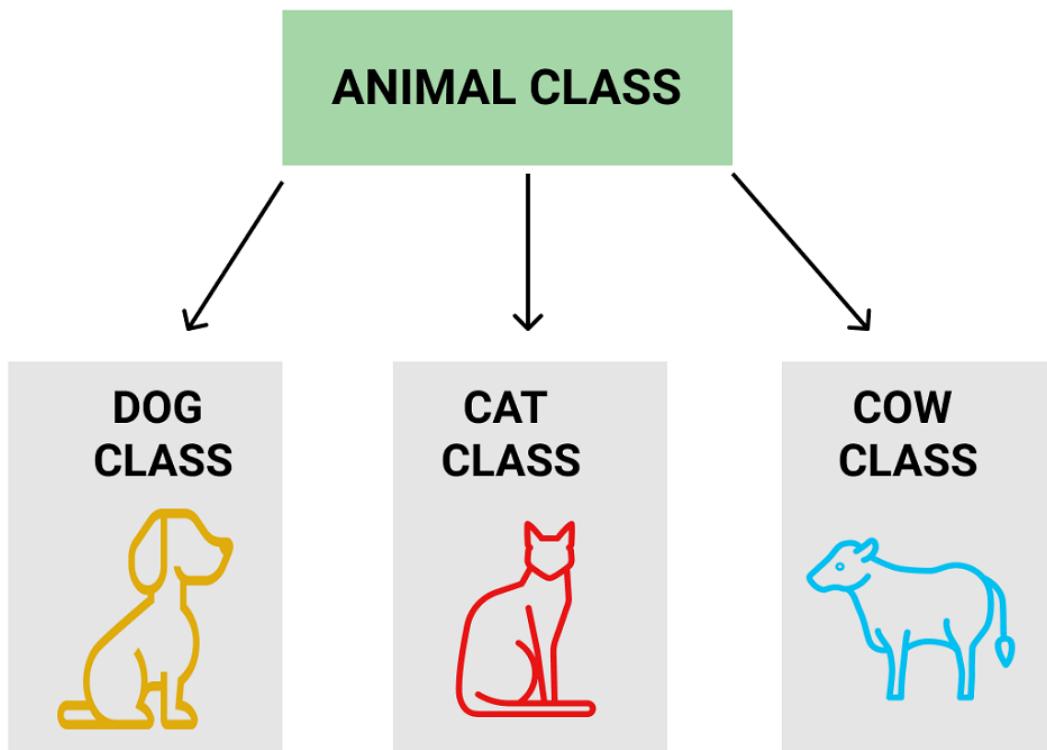
To know more about polymorphism, refer to this article – [Polymorphism in C++](#)

## Inheritance

The capability of a class to derive properties and characteristics from another class is called Inheritance. Inheritance is one of the most important features of Object-Oriented Programming.

- **Sub Class:** The class that inherits properties from another class is called Sub class or Derived Class.
- **Super Class:** The class whose properties are inherited by a sub-class is called Base Class or Superclass.
- **Reusability:** Inheritance supports the concept of "reusability", i.e. when we want to create a new class and there is already a class that includes some of the code that we want, we can derive our new class from the existing class. By doing this, we are reusing the fields and methods of the existing class.

**Example:** Dog, Cat, Cow can be Derived Class of Animal Base Class.



*Inheritance in C++*

To know more about Inheritance, refer to this article – [Inheritance in C++](#)

## Dynamic Binding

In dynamic binding, the code to be executed in response to the function call is decided at runtime. C++ has virtual functions to support this. Because dynamic binding is flexible, it avoids the drawbacks of static binding, which connected the function call and definition at build time.

**Example:**

## C++

```
// C++ Program to Demonstrate the Concept of Dynamic binding
// with the help of virtual function
#include <iostream>
using namespace std;

class GFG {
public:
    void call_Function() // function that call print
    {
        print();
    }
    void print() // the display function
    {
        cout << "Printing the Base class Content" << endl;
    }
};
class GFG2 : public GFG // GFG2 inherit a publicly
{
public:
    void print() // GFG2's display
    {
        cout << "Printing the Derived class Content"
            << endl;
    }
};
int main()
{
    GFG geeksforgeeks; // Creating GFG's pbject
    geeksforgeeks.call_Function(); // Calling call_Function
    GFG2 geeksforgeeks2; // creating GFG2 object
    geeksforgeeks2.call_Function(); // calling call_Function
                                // for GFG2 object
    return 0;
}
```

## Output

```
Printing the Base class Content
Printing the Base class Content
```

As we can see, the `print()` function of the parent class is called even from the derived class object. To resolve this we use virtual functions.

## Message Passing

Objects communicate with one another by sending and receiving information. A message for an object is a request for the execution of a procedure and therefore will invoke a function in the receiving object that generates the desired results. Message passing

involves specifying the name of the object, the name of the function, and the information to be sent.

## Related Articles:

- [Classes and Objects](#)
- [Inheritance](#)
- [Access Modifiers](#)
- [Abstraction](#)

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

1.45k

## Related Articles

1. Why C++ is partially Object Oriented Language?
2. OOPs | Object Oriented Design
3. Can a C++ class have an object of self type?
4. Object Slicing in C++
5. Preventing Object Copy in C++ (3 Different Ways)
6. Where is an object stored if it is created inside a block in C++?
7. cerr - Standard Error Stream Object in C++
8. How to add reference of an object in Container Classes
9. Dynamic initialization of object in C++
10. Object Delegation in C++

Previous

Next

**SALE!**  
**GeeksforGeeks Courses Upto 25% Off Enroll Now!**



Save 25% on Courses DSA Data Structures Algorithms Interview Preparation Data Science T

# C++ Classes and Objects

Difficulty Level : Easy • Last Updated : 16 Feb, 2023

Read

Discuss(20+)

Courses

Practice

Video

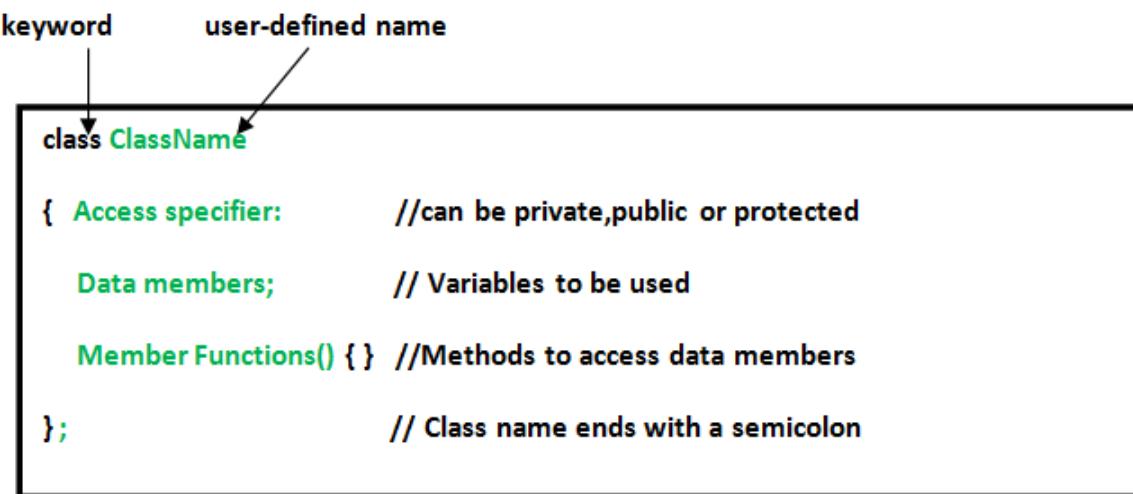
**Class:** A class in C++ is the building block that leads to Object-Oriented programming. It is a user-defined data type, which holds its own data members and member functions, which can be accessed and used by creating an instance of that class. A C++ class is like a blueprint for an object. For Example: Consider the Class of **Cars**. There may be many cars with different names and brand but all of them will share some common properties like all of them will have *4 wheels, Speed Limit, Mileage range* etc. So here, Car is the class and wheels, speed limits, mileage are their properties.

- A Class is a user defined data-type which has data members and member functions.
- Data members are the data variables and member functions are the functions used to manipulate these variables and together these data members and member functions defines the properties and behavior of the objects in a Class.
- In the above example of class *Car*, the data member will be *speed limit, mileage* etc and member functions can be *apply brakes, increase speed* etc.

An **Object** is an instance of a Class. When a class is defined, no memory is allocated but when it is instantiated (i.e. an object is created) memory is allocated.

## Defining Class and Declaring Objects

A class is defined in C++ using keyword `class` followed by the name of class. The body of class is defined inside the curly brackets and terminated by a semicolon at the end.



**Declaring Objects:** When a class is defined, only the specification for the object is defined; no memory or storage is allocated. To use the data and access functions defined in the class, you need to create objects. **Syntax:**

AD

**ClassName ObjectName;**

**Accessing data members and member functions:** The data members and member functions of class can be accessed using the dot('.') operator with the object. For example if the name of object is *obj* and you want to access the member function with the name *printName()* then you will have to write *obj.printName()* .

### Accessing Data Members

The public data members are also accessed in the same way given however the private data members are not allowed to be accessed directly by the object. Accessing a data member depends solely on the access control of that data member. This access control is given by [Access modifiers in C++](#). There are three access modifiers : **public, private and protected**.

## CPP

```
// C++ program to demonstrate accessing of data members
#include <bits/stdc++.h>
using namespace std;
```

```

class Geeks {
    // Access specifier
public:
    // Data Members
    string geekname;
    // Member Functions()
    void printname() { cout << "Geekname is:" << geekname; }
};

int main()
{
    // Declare an object of class geeks
    Geeks obj1;
    // accessing data member
    obj1.geekname = "Abhi";
    // accessing member function
    obj1.printname();
    return 0;
}

```

## Output

Geekname is:Abhi

## Member Functions in Classes

There are 2 ways to define a member function:

- Inside class definition
- Outside class definition

To define a member function outside the class definition we have to use the scope resolution :: operator along with class name and function name.

## CPP

```

// C++ program to demonstrate function
// declaration outside class

#include <bits/stdc++.h>
using namespace std;
class Geeks
{
    public:
    string geekname;
    int id;

    // printname is not defined inside class definition
    void printname();

    // printid is defined inside class definition
    void printid()
    {

```

```

        cout << "Geek id is: " << id;
    }
};

// Definition of printname using scope resolution operator ::

void Geeks::printname()
{
    cout << "Geekname is: " << geekname;
}
int main() {

    Geeks obj1;
    obj1.geekname = "xyz";
    obj1.id=15;

    // call printname()
    obj1.printname();
    cout << endl;

    // call printid()
    obj1.printid();
    return 0;
}

```

## Output:

```

Geekname is: xyz
Geek id is: 15

```

Note that all the member functions defined inside the class definition are by default **inline**, but you can also make any non-class function inline by using keyword `inline` with them. Inline functions are actual functions, which are copied everywhere during compilation, like pre-processor macro, so the overhead of function calling is reduced. Note: Declaring a [friend function](#) is a way to give private access to a non-member function.

## Constructors

Constructors are special class members which are called by the compiler every time an object of that class is instantiated. Constructors have the same name as the class and may be defined inside or outside the class definition. There are 3 types of constructors:

- [Default constructors](#)
- Parameterized constructors
- [Copy constructors](#)

## CPP

```
// C++ program to demonstrate constructors

#include <bits/stdc++.h>
using namespace std;
class Geeks
{
public:
    int id;

    //Default Constructor
    Geeks()
    {
        cout << "Default Constructor called" << endl;
        id=-1;
    }

    //Parameterized Constructor
    Geeks(int x)
    {
        cout << "Parameterized Constructor called " << endl;
        id=x;
    }
};

int main() {

    // obj1 will call Default Constructor
    Geeks obj1;
    cout << "Geek id is: " << obj1.id << endl;

    // obj2 will call Parameterized Constructor
    Geeks obj2(21);
    cout << "Geek id is: " << obj2.id << endl;
    return 0;
}
```

### Output:

```
Default Constructor called
Geek id is: -1
Parameterized Constructor called
Geek id is: 21
```

A **Copy Constructor** creates a new object, which is exact copy of the existing object. The compiler provides a default Copy Constructor to all the classes. Syntax:

```
class-name (class-name &){}
```

## Constructors

Destructor is another special member function that is called by the compiler when the scope of the object ends.

---

## CPP

```
// C++ program to explain destructors

#include <bits/stdc++.h>
using namespace std;
class Geeks
{
public:
    int id;

    //Definition for Destructor
    ~Geeks()
    {
        cout << "Destructor called for id: " << id << endl;
    }
};

int main()
{
    Geeks obj1;
    obj1.id=7;
    int i = 0;
    while ( i < 5 )
    {
        Geeks obj2;
        obj2.id=i;
        i++;
    } // Scope for obj2 ends here

    return 0;
} // Scope for obj1 ends here
```

### Output:

```
Destructor called for id: 0
Destructor called for id: 1
Destructor called for id: 2
Destructor called for id: 3
Destructor called for id: 4
Destructor called for id: 7
```

### Interesting Fact (Rare Known concept)

Why do we give semicolon at the end of class ?

Many people might say that its a basic syntax and we should give semicolon at end of class as its rule define in cpp . But the main reason why semi-colons is there at end of class is compiler checks if user is trying to create an instance of class at the end of it .

Yes just like structure , Union we can also create the instance of class at the end just before the semicolon. As a result once execution reaches at that line it creates class and allocates memory to your instance

---

## C++

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

class Demo{
    int a, b;
public:
    Demo() // default constructor
    {
        cout << "Default Constructor" << endl;
    }
    Demo(int a, int b):a(a),b(b) //parameterised constructor
    {
        cout << "parameterized constructor -values" << a << " " << b << endl;
    }
}

instance;

int main() {

    return 0;
}
```

## Output

Default Constructor

We can see that we have created class instance of Demo with name "instance" , as a result output we can see is Default Constructor is called.

Similarly we can also call the parameterized constructor just by passing values here

---

## C++

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

class Demo{
public:
```

```

int a, b;
Demo()
{
    cout << "Default Constructor" << endl;
}
Demo(int a, int b):a(a),b(b)
{
    cout << "parameterized Constructor values-" << a << " " << b << endl;
}

}instance(100,200);

int main() {

    return 0;
}

```

## Output

parameterized Constructor values-100 200

So by creating instance just before the semicolon , we can create the Instance of class

### Pure Virtual Destructor **Related Articles:**

- [Multiple Inheritance in C++](#)
- [C++ Quiz](#)

This article is contributed by **Abhirav Kariya**. If you like GeeksforGeeks and would like to contribute, you can also write an article using write.geeksforgeeks.org or mail your article to review-team@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

771

## Related Articles

1. [Classes and Objects in Java](#)
2. [Catching Base and Derived Classes as Exceptions in C++ and Java](#)
3. [Pure Virtual Functions and Abstract Classes in C++](#)

**SALE!**  
**GeeksforGeeks Courses Upto 25% Off Enroll Now!**



Save 25% on Courses DSA Data Structures Algorithms Interview Preparation Data Science T

# Inheritance in C++

Difficulty Level : Easy • Last Updated : 17 Feb, 2023

[Read](#) [Discuss\(90+\)](#) [Courses](#) [Practice](#) [Video](#)

The capability of a [class](#) to derive properties and characteristics from another class is called **Inheritance**. Inheritance is one of the most important features of Object-Oriented Programming.

Inheritance is a feature or a process in which, new classes are created from the existing classes. The new class created is called "derived class" or "child class" and the existing class is known as the "base class" or "parent class". The derived class now is said to be inherited from the base class.

When we say derived class inherits the base class, it means, the derived class inherits all the properties of the base class, without changing the properties of base class and may add new features to its own. These new features in the derived class will not affect the base class. The derived class is the specialized class for the base class.

- **Sub Class:** The class that inherits properties from another class is called Subclass or Derived Class.
- **Super Class:** The class whose properties are inherited by a subclass is called Base Class or Superclass.

**The article is divided into the following subtopics:**

AD

- Why and when to use inheritance?
- Modes of Inheritance
- Types of Inheritance

## Why and when to use inheritance?

Consider a group of vehicles. You need to create classes for Bus, Car, and Truck. The methods fuelAmount(), capacity(), applyBrakes() will be the same for all three classes. If we create these classes avoiding inheritance then we have to write all of these functions in each of the three classes as shown below figure:

**Class Bus**

**Class Car**

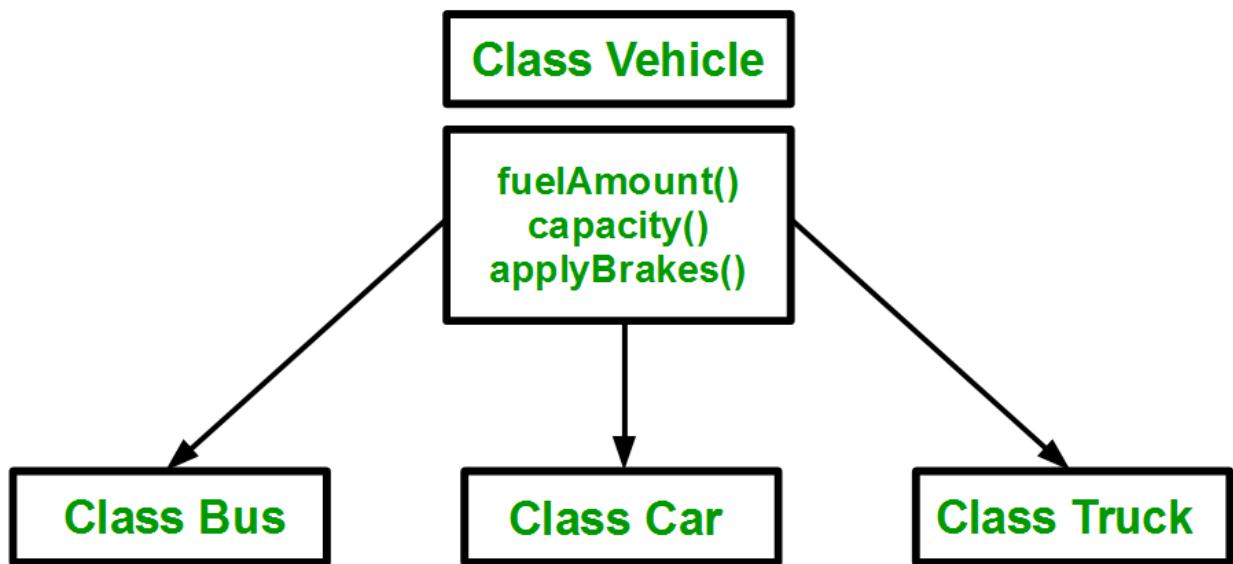
**Class Truck**

**fuelAmount()  
capacity()  
applyBrakes()**

**fuelAmount()  
capacity()  
applyBrakes()**

**fuelAmount()  
capacity()  
applyBrakes()**

You can clearly see that the above process results in duplication of the same code 3 times. This increases the chances of error and data redundancy. To avoid this type of situation, inheritance is used. If we create a class Vehicle and write these three functions in it and inherit the rest of the classes from the vehicle class, then we can simply avoid the duplication of data and increase re-usability. Look at the below diagram in which the three classes are inherited from vehicle class:



Using inheritance, we have to write the functions only one time instead of three times as we have inherited the rest of the three classes from the base class (Vehicle).

**Implementing inheritance in C++:** For creating a sub-class that is inherited from the base class we have to follow the below syntax.

**Derived Classes:** A Derived class is defined as the class derived from the base class.

#### Syntax:

```

class <derived_class_name> : <access-specifier> <base_class_name>
{
    //body
}
  
```

Where

class – keyword to create a new class

derived\_class\_name – name of the new class, which will inherit the base class

access-specifier – either of private, public or protected. If neither is specified, PRIVATE is taken as default

base-class-name – name of the base class

**Note:** A derived class doesn't inherit **access** to private data members. However, it does inherit a full parent object, which contains any private members which that class declares.

#### Example:

1. class ABC : private XYZ //private derivation  
 { }
2. class ABC : public XYZ //public derivation  
 { }
3. class ABC : protected XYZ //protected derivation  
 { }

```
4. class ABC: XYZ           //private derivation by default
{ }
```

**Note:**

- o When a base class is privately inherited by the derived class, public members of the base class becomes the private members of the derived class and therefore, the public members of the base class can only be accessed by the member functions of the derived class. They are inaccessible to the objects of the derived class.
  - o On the other hand, when the base class is publicly inherited by the derived class, public members of the base class also become the public members of the derived class. Therefore, the public members of the base class are accessible by the objects of the derived class as well as by the member functions of the derived class.
- 

**C++**

```
// Example: define member function without argument within the class

#include<iostream>
using namespace std;

class Person
{
    int id;
    char name[100];

public:
    void set_p()
    {
        cout<<"Enter the Id:" ;
        cin>>id;
        fflush(stdin);
        cout<<"Enter the Name:" ;
        cin.get(name,100);
    }

    void display_p()
    {
        cout<<endl<<id<<"\t"<<name<<"\t";
    }
};

class Student: private Person
{
    char course[50];
    int fee;

public:
    void set_s()
    {
```

```

    set_p();
    cout<<"Enter the Course Name:";
    fflush(stdin);
    cin.getline(course,50);
    cout<<"Enter the Course Fee:";
    cin>>fee;
}

void display_s()
{
    display_p();
    cout<<course<<"\t"<<fee<<endl;
}
};

main()
{
    Student s;
    s.set_s();
    s.display_s();
    return 0;
}

```

## Output:

```

Enter the Id: 101
Enter the Name: Dev
Enter the Course Name: GCS
Enter the Course Fee:70000

```

---

101	Dev	GCS	70000
-----	-----	-----	-------

---

## C++

```

// Example: define member function without argument outside the class

#include<iostream>
using namespace std;

class Person
{
    int id;
    char name[100];

public:
    void set_p();
    void display_p();
};

void Person::set_p()
{

```

```
cout<<"Enter the Id:";  
cin>>id;  
fflush(stdin);  
cout<<"Enter the Name:";  
cin.get(name,100);  
}  
  
void Person::display_p()  
{  
    cout<<endl<<id<<"\t"<<name;  
}  
  
class Student: private Person  
{  
    char course[50];  
    int fee;  
  
    public:  
        void set_s();  
        void display_s();  
};  
  
void Student::set_s()  
{  
    set_p();  
    cout<<"Enter the Course Name:";  
fflush(stdin);  
    cin.getline(course,50);  
    cout<<"Enter the Course Fee:";  
    cin>>fee;  
}  
  
void Student::display_s()  
{  
    display_p();  
    cout<<"\t"<<course<<"\t"<<fee;  
}  
  
main()  
{  
    Student s;  
    s.set_s();  
    s.display_s();  
    return 0;  
}
```

## Output

```
Enter the Id:Enter the Name:Enter the Course Name:Enter the Course Fee:  
0      t      0
```

## C++

```
// Example: define member function with argument outside the class

#include<iostream>
#include<string.h>
using namespace std;

class Person
{
    int id;
    char name[100];

public:
    void set_p(int,char[]);
    void display_p();
};

void Person::set_p(int id,char n[])
{
    this->id=id;
    strcpy(this->name,n);
}

void Person::display_p()
{
    cout<<endl<<id<<"\t"<<name;
}

class Student: private Person
{
    char course[50];
    int fee;
public:
    void set_s(int,char[],char[],int);
    void display_s();
};

void Student::set_s(int id,char n[],char c[],int f)
{
    set_p(id,n);
    strcpy(course,c);
    fee=f;
}

void Student::display_s()
{
    display_p();
    cout<<"\t"<<course<<"\t"<<fee;
}

main()
{
    Student s;
```

```
s.set_s(1001,"Ram","B.Tech",2000);
s.display_s();
return 0;
}
```

---

## CPP

```
// C++ program to demonstrate implementation
// of Inheritance

#include <bits/stdc++.h>
using namespace std;

// Base class
class Parent {
public:
    int id_p;
};

// Sub class inheriting from Base Class(Parent)
class Child : public Parent {
public:
    int id_c;
};

// main function
int main()
{
    Child obj1;

    // An object of class child has all data members
    // and member functions of class parent
    obj1.id_c = 7;
    obj1.id_p = 91;
    cout << "Child id is: " << obj1.id_c << '\n';
    cout << "Parent id is: " << obj1.id_p << '\n';

    return 0;
}
```

## Output

```
Child id is: 7
Parent id is: 91
```

## Output:

```
Child id is: 7
Parent id is: 91
```

In the above program, the 'Child' class is publicly inherited from the 'Parent' class so the public data members of the class 'Parent' will also be inherited by the class 'Child'.

**Modes of Inheritance:** There are 3 modes of inheritance.

1. **Public Mode:** If we derive a subclass from a public base class. Then the public member of the base class will become public in the derived class and protected members of the base class will become protected in the derived class.
2. **Protected Mode:** If we derive a subclass from a Protected base class. Then both public members and protected members of the base class will become protected in the derived class.
3. **Private Mode:** If we derive a subclass from a Private base class. Then both public members and protected members of the base class will become Private in the derived class.

**Note:** The private members in the base class cannot be directly accessed in the derived class, while protected members can be directly accessed. For example, Classes B, C, and D all contain the variables x, y, and z in the below example. It is just a question of access.

## CPP

```
// C++ Implementation to show that a derived class
// doesn't inherit access to private data members.
// However, it does inherit a full parent object.
class A {
public:
    int x;

protected:
    int y;

private:
    int z;
};

class B : public A {
    // x is public
    // y is protected
    // z is not accessible from B
};

class C : protected A {
    // x is protected
    // y is protected
    // z is not accessible from C
};
```

```
class D : private A // 'private' is default for classes
{
    // x is private
    // y is private
    // z is not accessible from D
};
```

The below table summarizes the above three modes and shows the access specifier of the members of the base class in the subclass when derived in public, protected and private modes:

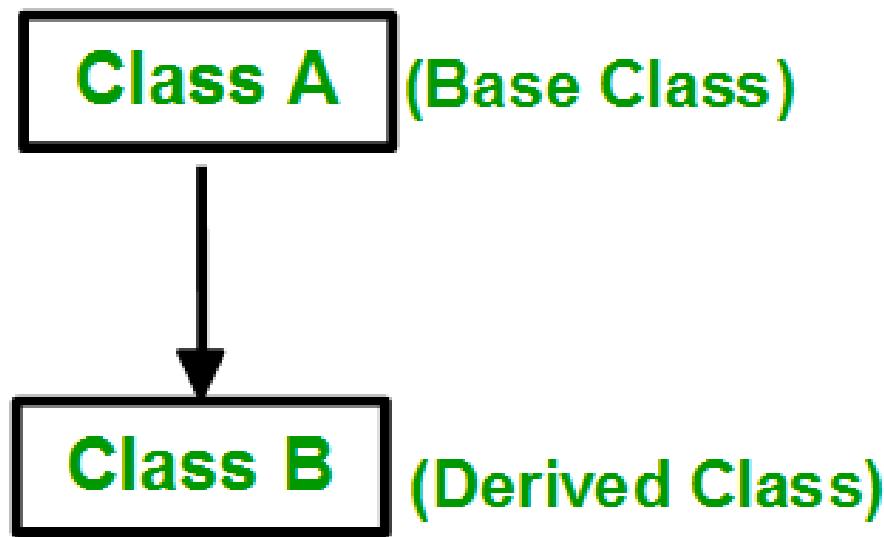
Base class member access specifier	Type of Inheritance		
	Public	Protected	Private
Public	Public	Protected	Private
Protected	Protected	Protected	Private
Private	Not accessible (Hidden)	Not accessible (Hidden)	Not accessible (Hidden)

## Types Of Inheritance:-

1. Single inheritance
2. Multilevel inheritance
3. Multiple inheritance
4. Hierarchical inheritance
5. Hybrid inheritance

## **Types of Inheritance in C++**

**1. Single Inheritance:** In single inheritance, a class is allowed to inherit from only one class. i.e. one subclass is inherited by one base class only.



### Syntax:

```
class subclass_name : access_mode base_class
{
    // body of subclass
};
```

OR

```
class A
{
    ...
};

class B: public A
{
    ...
};
```

---

### CPP

```
// C++ program to explain
// Single inheritance
#include<iostream>
using namespace std;

// base class
class Vehicle {
public:
    Vehicle()
    {
        cout << "This is a Vehicle\n";
    }
}
```

```

    }
};

// sub class derived from a single base classes
class Car : public Vehicle {

};

// main function
int main()
{
    // Creating object of sub class will
    // invoke the constructor of base classes
    Car obj;
    return 0;
}

```

## Output

This is a Vehicle

---

## C++

```

// Example:

#include<iostream>
using namespace std;

class A
{
protected:
    int a;

public:
    void set_A()
    {
        cout<<"Enter the Value of A=";
        cin>>a;

    }
    void disp_A()
    {
        cout<<endl<<"Value of A="<<a;
    }
};

class B: public A
{
    int b,p;

```

```

public:
    void set_B()
    {
        set_A();
        cout<<"Enter the Value of B=";
        cin>>b;
    }

    void disp_B()
    {
        disp_A();
        cout<<endl<<"Value of B="<<b;
    }

    void cal_product()
    {
        p=a*b;
        cout<<endl<<"Product of "<<a<<" * "<<b<<" = "<<p;
    }

};

main()
{
    B _b;
    _b.set_B();
    _b.cal_product();

    return 0;
}

```

Output:- Enter the Value of A= 3 3 Enter the Value of B= 5 5 Product of 3 \* 5 = 15

---

## C++

```

// Example:

#include<iostream>
using namespace std;

class A
{
    protected:
    int a;

    public:
        void set_A(int x)
        {
            a=x;
        }
}

```

```

void disp_A()
{
    cout<<endl<<"Value of A="<<a;
}
};

class B: public A
{
    int b,p;

    public:
        void set_B(int x,int y)
        {
            set_A(x);
            b=y;
        }

        void disp_B()
        {
            disp_A();
            cout<<endl<<"Value of B="<<b;
        }

        void cal_product()
        {
            p=a*b;
            cout<<endl<<"Product of "<<a<<" * "<<b<<" = "<<p;
        }
}

main()
{
    B _b;
    _b.set_B(4,5);
    _b.cal_product();

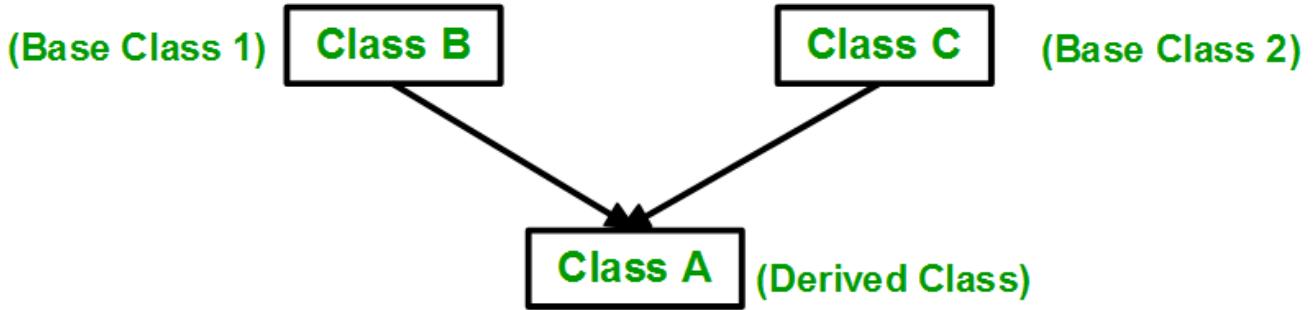
    return 0;
}

```

## Output

Product of 4 \* 5 = 20

**2. Multiple Inheritance:** Multiple Inheritance is a feature of C++ where a class can inherit from more than one class. i.e one **subclass** is inherited from more than one **base class**.



### Syntax:

```

class subclass_name : access_mode base_class1, access_mode base_class2, ....
{
    // body of subclass
};
  
```

```

class B
{
...
};

class C
{
...
};

class A: public B, public C
{
...
};
```

Here, the number of base classes will be separated by a comma (',') and the access mode for every base class must be specified.

### CPP

```

// C++ program to explain
// multiple inheritance
#include <iostream>
```

```

using namespace std;

// first base class
class Vehicle {
public:
    Vehicle() { cout << "This is a Vehicle\n"; }
};

// second base class
class FourWheeler {
public:
    FourWheeler()
    {
        cout << "This is a 4 wheeler Vehicle\n";
    }
};

// sub class derived from two base classes
class Car : public Vehicle, public FourWheeler {
};

// main function
int main()
{
    // Creating object of sub class will
    // invoke the constructor of base classes.
    Car obj;
    return 0;
}

```

## Output

This is a Vehicle  
This is a 4 wheeler Vehicle

---

## C++

```

// Example:

#include<iostream>
using namespace std;

class A
{
    protected:
    int a;

    public:
        void set_A()
    {
        cout<<"Enter the Value of A=";
    }
}

```

```
        cin>>a;  
    }  
  
    void disp_A()  
    {  
        cout<<endl<<"Value of A="<<a;  
    }  
};  
  
class B: public A  
{  
protected:  
    int b;  
  
public:  
    void set_B()  
    {  
        cout<<"Enter the Value of B=";  
        cin>>b;  
    }  
  
    void disp_B()  
    {  
        cout<<endl<<"Value of B="<<b;  
    }  
};  
  
class C: public B  
{  
int c,p;  
  
public:  
    void set_C()  
    {  
        cout<<"Enter the Value of C=";  
        cin>>c;  
    }  
  
    void disp_C()  
    {  
        cout<<endl<<"Value of C="<<c;  
    }  
  
    void cal_product()  
    {  
        p=a*b*c;  
        cout<<endl<<"Product of "<<a<<" * "<<b<<" * "<<c<<" = "<<p;  
    }  
};  
  
main()  
{  
  
    C _c;  
    _c.set_A();
```

```

_c.set_B();
_c.set_C();
_c.disp_A();
_c.disp_B();
_c.disp_C();
_c.cal_product();

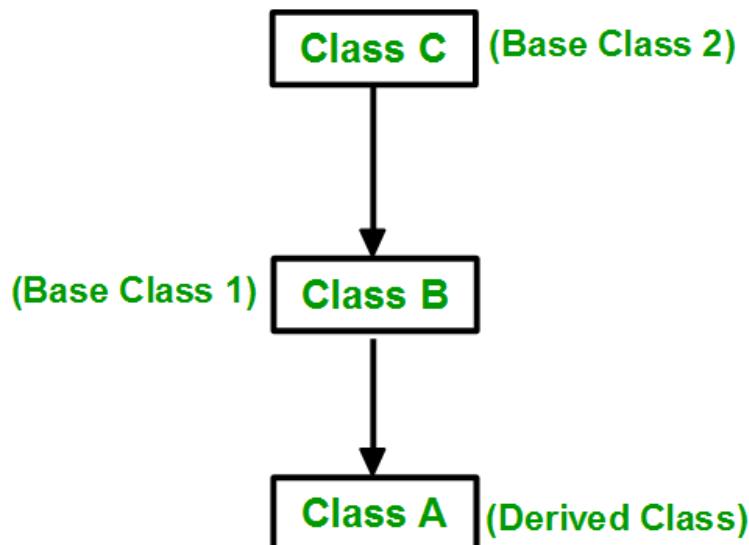
return 0;

}

```

To know more about it, please refer to the article [Multiple Inheritances](#).

**3. Multilevel Inheritance:** In this type of inheritance, a derived class is created from another derived class.



Syntax:-

```

class C
{
...
};

class B:public C
{
...
};

class A: public B
{
...
};

```

## CPP

```
// C++ program to implement
// Multilevel Inheritance
#include <iostream>
using namespace std;

// base class
class Vehicle {
public:
    Vehicle() { cout << "This is a Vehicle\n"; }
};

// first sub_class derived from class vehicle
class fourWheeler : public Vehicle {
public:
    fourWheeler()
    {
        cout << "Objects with 4 wheels are vehicles\n";
    }
};

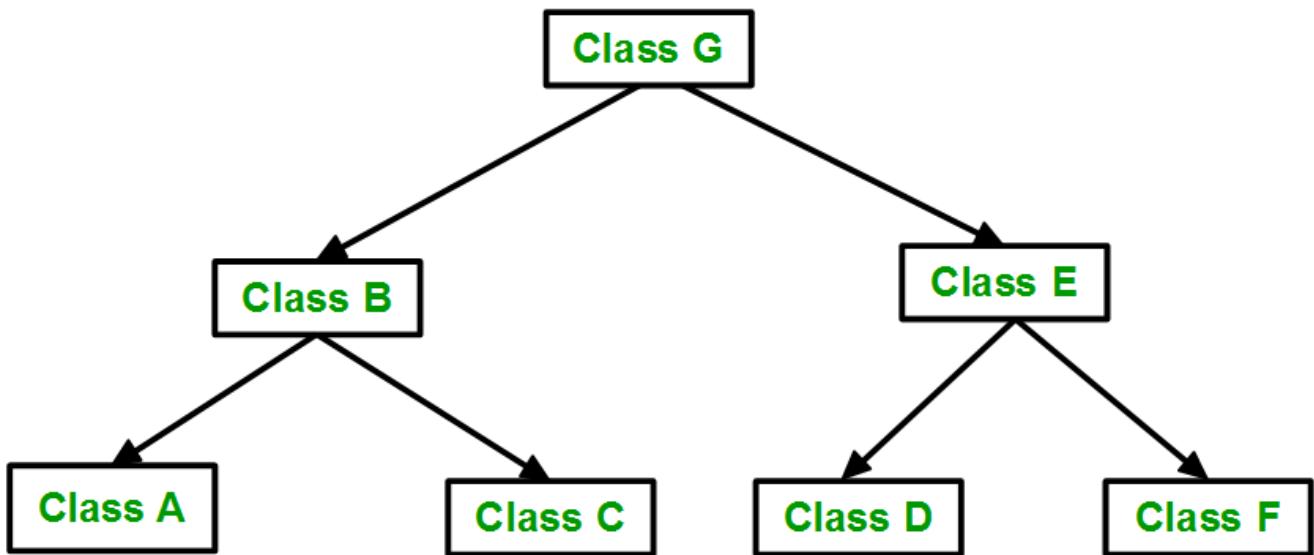
// sub class derived from the derived base class fourWheeler
class Car : public fourWheeler {
public:
    Car() { cout << "Car has 4 Wheels\n"; }
};

// main function
int main()
{
    // Creating object of sub class will
    // invoke the constructor of base classes.
    Car obj;
    return 0;
}
```

## Output

```
This is a Vehicle
Objects with 4 wheels are vehicles
Car has 4 Wheels
```

**4. Hierarchical Inheritance:** In this type of inheritance, more than one subclass is inherited from a single base class. i.e. more than one derived class is created from a single base class.



Syntax:-

```

class A
{
    // body of the class A.
}

class B : public A
{
    // body of class B.
}

class C : public A
{
    // body of class C.
}

class D : public A
{
    // body of class D.
}
  
```

## CPP

```

// C++ program to implement
// Hierarchical Inheritance
#include <iostream>
using namespace std;

// base class
class Vehicle {
public:
    Vehicle() { cout << "This is a Vehicle\n"; }
};
  
```

```
// first sub class
class Car : public Vehicle {
};

// second sub class
class Bus : public Vehicle {
};

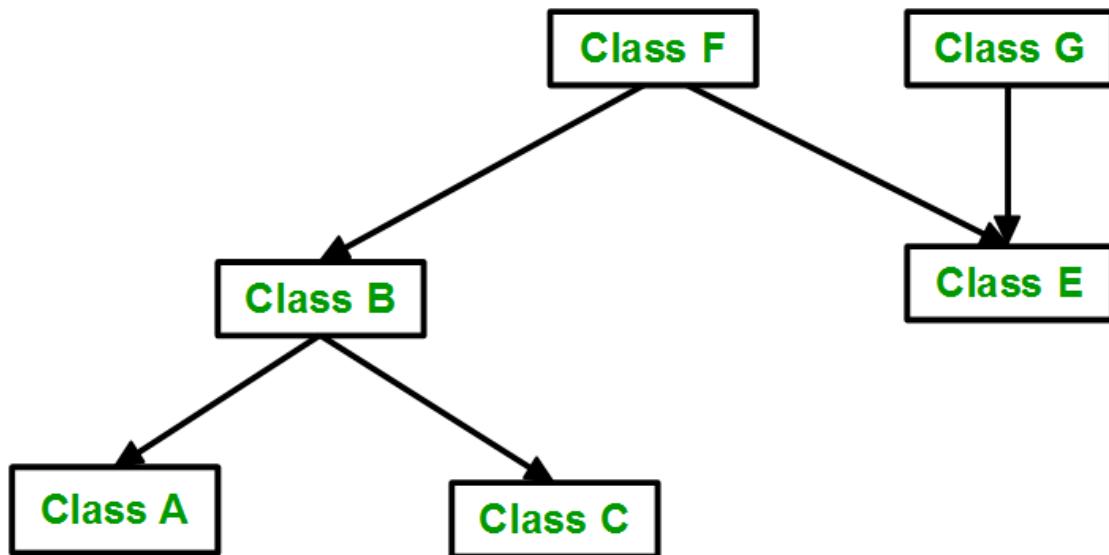
// main function
int main()
{
    // Creating object of sub class will
    // invoke the constructor of base class.
    Car obj1;
    Bus obj2;
    return 0;
}
```

## Output

This is a Vehicle  
This is a Vehicle

**5. Hybrid (Virtual) Inheritance:** Hybrid Inheritance is implemented by combining more than one type of inheritance. For example: Combining Hierarchical inheritance and Multiple Inheritance.

Below image shows the combination of hierarchical and multiple inheritances:



```
// C++ program for Hybrid Inheritance

#include <iostream>
using namespace std;

// base class
class Vehicle {
public:
    Vehicle() { cout << "This is a Vehicle\n"; }
};

// base class
class Fare {
public:
    Fare() { cout << "Fare of Vehicle\n"; }
};

// first sub class
class Car : public Vehicle {
};

// second sub class
class Bus : public Vehicle, public Fare {
};

// main function
int main()
{
    // Creating object of sub class will
    // invoke the constructor of base class.
    Bus obj2;
    return 0;
}
```

## Output

This is a Vehicle  
 Fare of Vehicle

## C++

```
// Example:

#include <iostream>
using namespace std;

class A
{
protected:
    int a;
```

```
public:  
void get_a()  
{  
    cout << "Enter the value of 'a' : ";  
    cin>>a;  
}  
};  
  
class B : public A  
{  
protected:  
int b;  
public:  
void get_b()  
{  
    cout << "Enter the value of 'b' : ";  
    cin>>b;  
}  
};  
class C  
{  
protected:  
int c;  
public:  
void get_c()  
{  
    cout << "Enter the value of c is : ";  
    cin>>c;  
}  
};  
  
class D : public B, public C  
{  
protected:  
int d;  
public:  
void mul()  
{  
    get_a();  
    get_b();  
    get_c();  
    cout << "Multiplication of a,b,c is : " <<a*b*c;  
}  
};  
  
int main()  
{  
D d;  
d.mul();  
return 0;  
}
```

## 6. A special case of hybrid inheritance: Multipath inheritance:

A derived class with two base classes and these two base classes have one common base class is called multipath inheritance. Ambiguity can arise in this type of inheritance.

### Example:

---

#### CPP

```
// C++ program demonstrating ambiguity in Multipath
// Inheritance

#include <iostream>
using namespace std;

class ClassA {
public:
    int a;
};

class ClassB : public ClassA {
public:
    int b;
};

class ClassC : public ClassA {
public:
    int c;
};

class ClassD : public ClassB, public ClassC {
public:
    int d;
};

int main()
{
    ClassD obj;

    // obj.a = 10;                      // Statement 1, Error
    // obj.a = 100;                     // Statement 2, Error

    obj.ClassB::a = 10; // Statement 3
    obj.ClassC::a = 100; // Statement 4

    obj.b = 20;
    obj.c = 30;
    obj.d = 40;

    cout << " a from ClassB : " << obj.ClassB::a;
    cout << "\n a from ClassC : " << obj.ClassC::a;

    cout << "\n b : " << obj.b;
    cout << "\n c : " << obj.c;
    cout << "\n d : " << obj.d << '\n';
}
```

```

    }
}

```

## Output

```

a from ClassB : 10
a from ClassC : 100
b : 20
c : 30
d : 40

```

## Output:

```

a from ClassB : 10
a from ClassC : 100
b : 20
c : 30
d : 40

```

In the above example, both ClassB and ClassC inherit ClassA, they both have a single copy of ClassA. However Class-D inherits both ClassB and ClassC, therefore Class-D has two copies of ClassA, one from ClassB and another from ClassC.

If we need to access the data member of ClassA through the object of Class-D, we must specify the path from which a will be accessed, whether it is from ClassB or ClassC, bcoz compiler can't differentiate between two copies of ClassA in Class-D.

## There are 2 Ways to Avoid this Ambiguity:

**1) Avoiding ambiguity using the scope resolution operator:** Using the scope resolution operator we can manually specify the path from which data member a will be accessed, as shown in statements 3 and 4, in the above example.

---

## CPP

```

obj.ClassB::a = 10;      // Statement 3
obj.ClassC::a = 100;     // Statement 4

```

**Note:** Still, there are two copies of ClassA in Class-D.

## 2) Avoiding ambiguity using the virtual base class:

---

## CPP

```
#include<iostream>

class ClassA
{
    public:
        int a;
};

class ClassB : virtual public ClassA
{
    public:
        int b;
};

class ClassC : virtual public ClassA
{
    public:
        int c;
};

class ClassD : public ClassB, public ClassC
{
    public:
        int d;
};

int main()
{
    ClassD obj;

    obj.a = 10;           // Statement 3
    obj.a = 100;          // Statement 4

    obj.b = 20;
    obj.c = 30;
    obj.d = 40;

    cout << "\n a : " << obj.a;
    cout << "\n b : " << obj.b;
    cout << "\n c : " << obj.c;
    cout << "\n d : " << obj.d << '\n';
}
```

## Output:

```
a : 100
b : 20
c : 30
d : 40
```

According to the above example, Class-D has only one copy of ClassA, therefore, statement 4 will overwrite the value of a, given in statement 3.

This article is contributed by **Harsh Agarwal**. If you like GeeksforGeeks and would like to contribute, you can also write an article using [write.geeksforgeeks.org](https://write.geeksforgeeks.org) or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

977

## Related Articles

1. Inheritance and Friendship in C++

---
2. Inheritance and Constructors in Java

---
3. Multiple Inheritance in C++

---
4. Does overloading work with Inheritance?

---
5. OOP in Python | Set 3 (Inheritance, examples of object, issubclass and super)

---
6. Java and Multiple Inheritance

---
7. Difference between Containment and Inheritance in C++

---
8. Difference between Single and Multiple Inheritance in C++

---
9. Difference between Inheritance and Polymorphism

---
10. Runtime Polymorphism in various types of Inheritance in C++

---

Previous

Next

### Article Contributed By :



GeeksforGeeks

### Vote for difficulty

SALE!  
GeeksforGeeks Courses Upto 25% Off Enroll Now!



Save 25% on Courses DSA Data Structures Algorithms Interview Preparation Data Science T

# C++ Polymorphism

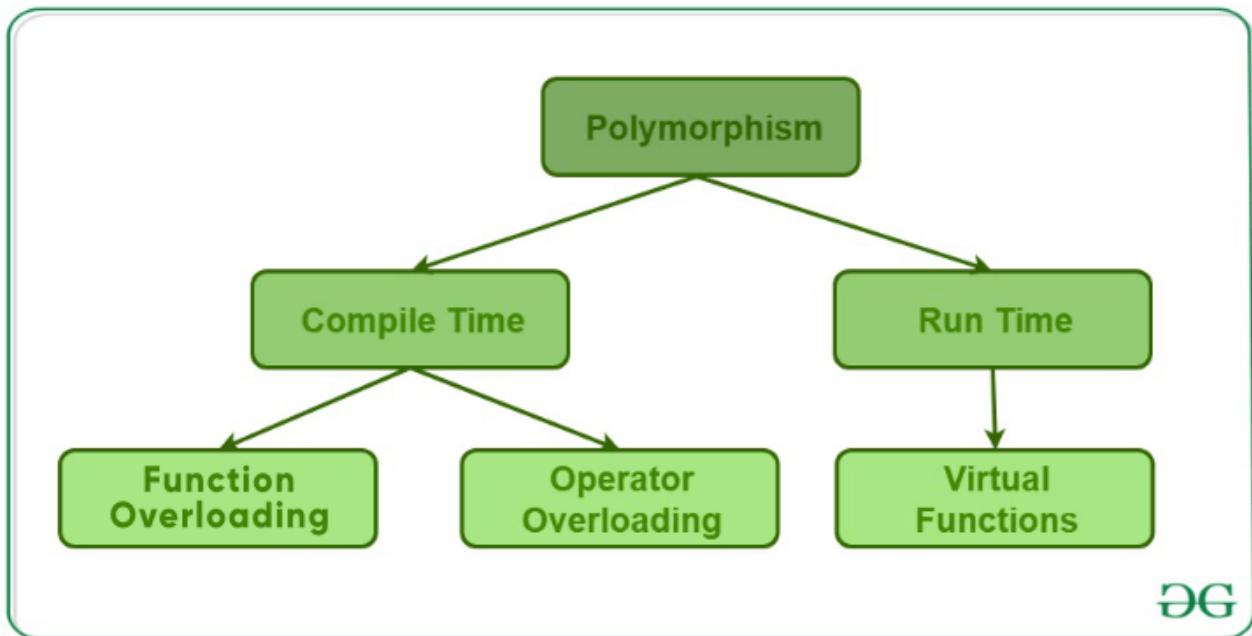
Difficulty Level : Easy • Last Updated : 03 Apr, 2023

[Read](#) [Discuss\(30+\)](#) [Courses](#) [Practice](#) [Video](#)

The word “polymorphism” means having many forms. In simple words, we can define polymorphism as the ability of a message to be displayed in more than one form. A real-life example of polymorphism is a person who at the same time can have different characteristics. A man at the same time is a father, a husband, and an employee. So the same person exhibits different behavior in different situations. This is called polymorphism. Polymorphism is considered one of the important features of Object-Oriented Programming.

## Types of Polymorphism

- **Compile-time Polymorphism.**
- **Runtime Polymorphism.**



# 1. Compile-Time Polymorphism

This type of polymorphism is achieved by function overloading or operator overloading.

## A. Function Overloading

When there are multiple functions with the same name but different parameters, then the functions are said to be **overloaded**, hence this is known as Function Overloading.

Functions can be overloaded by **changing the number of arguments** or/and **changing the type of arguments**. In simple terms, it is a feature of object-oriented programming providing many functions to have the same name but distinct parameters when numerous tasks are listed under one function name. There are certain [Rules of Function Overloading](#) that should be followed while overloading a function.

Below is the C++ program to show function overloading or compile-time polymorphism:

AD

---

## C++

```
// C++ program to demonstrate
// function overloading or
// Compile-time Polymorphism
#include <bits/stdc++.h>

using namespace std;
class Geeks {
public:

    // Function with 1 int parameter
    void func(int x)
    {
        cout << "value of x is " <<
            x << endl;
    }

    // Function with same name but
    // 1 double parameter
    void func(double x)
    {
```

```

cout << "value of x is " <<
      x << endl;
}

// Function with same name and
// 2 int parameters
void func(int x, int y)
{
    cout << "value of x and y is " <<
          x << ", " << y << endl;
}
};

// Driver code
int main()
{
    Geeks obj1;

    // Function being called depends
    // on the parameters passed
    // func() is called with int value
    obj1.func(7);

    // func() is called with double value
    obj1.func(9.132);

    // func() is called with 2 int values
    obj1.func(85, 64);
    return 0;
}

```

## Output

```

value of x is 7
value of x is 9.132
value of x and y is 85, 64

```

**Explanation:** In the above example, a single function named function **func()** acts differently in three different situations, which is a property of polymorphism.

To know more about this, you can refer to the article – [Function Overloading in C++](#).

## B. Operator Overloading

C++ has the ability to provide the operators with a special meaning for a data type, this ability is known as operator overloading. For example, we can make use of the addition operator (+) for string class to concatenate two strings. We know that the task of this operator is to add two operands. So a single operator '+', when placed between integer operands, adds them and when placed between string operands, concatenates them.

Below is the C++ program to demonstrate operator overloading:

## CPP

```
// C++ program to demonstrate
// Operator Overloading or
// Compile-Time Polymorphism
#include <iostream>
using namespace std;

class Complex {
private:
    int real, imag;

public:
    Complex(int r = 0,
            int i = 0)
    {
        real = r;
        imag = i;
    }

    // This is automatically called
    // when '+' is used with between
    // two Complex objects
    Complex operator+(Complex const& obj)
    {
        Complex res;
        res.real = real + obj.real;
        res.imag = imag + obj.imag;
        return res;
    }

    void print()
    {
        cout << real << " + i" <<
            imag << endl;
    }
};

// Driver code
int main()
{
    Complex c1(10, 5), c2(2, 4);

    // An example call to "operator+"
    Complex c3 = c1 + c2;
    c3.print();
}
```

## Output

12 + i9

**Explanation:** In the above example, the operator '+' is overloaded. Usually, this operator is used to add two numbers (integers or floating point numbers), but here the operator is made to perform the addition of two imaginary or complex numbers.

To know more about this one, refer to the article – [Operator Overloading](#).

## 2. Runtime Polymorphism

This type of polymorphism is achieved by **Function Overriding**. Late binding and dynamic polymorphism are other names for runtime polymorphism. The function call is resolved at runtime in [runtime polymorphism](#). In contrast, with compile time polymorphism, the compiler determines which function call to bind to the object after deducing it at runtime.

### A. Function Overriding

Function Overriding occurs when a derived class has a definition for one of the member functions of the base class. That base function is said to be overridden.

```
class Parent
{
public:
    void GeeksforGeeks()
    {
        statements;
    }
};

class Child: public Parent
{
public:
    void GeeksforGeeks() ←
    {
        Statements;
    }
};

int main()
{
    Child Child_Derived;
    Child_Derived.GeeksforGeeks(); →
    return 0;
}
```

*Function overriding Explanation*

Below is the C++ program to demonstrate function overriding:

---

**C++**

```
// C++ program for function overriding
#include <bits/stdc++.h>
using namespace std;

class base {
public:
    virtual void print()
    {
        cout << "print base class" <<
            endl;
    }

    void show()
    {
        cout << "show base class" <<
            endl;
    }
};

class derived : public base {
public:

    // print () is already virtual function in
    // derived class, we could also declared as
    // virtual void print () explicitly
    void print()
    {
        cout << "print derived class" <<
            endl;
    }

    void show()
    {
        cout << "show derived class" <<
            endl;
    }
};

// Driver code
int main()
{
    base* bptr;
    derived d;
    bptr = &d;

    // Virtual function, binded at
    // runtime (Runtime polymorphism)
    bptr->print();
```

```

// Non-virtual function, binded
// at compile time
bptr->show();

return 0;
}

```

## Output

```

print derived class
show base class

```

## Runtime Polymorphism with Data Members

Runtime Polymorphism can be achieved by data members in C++. Let's see an example where we are accessing the field by reference variable which refers to the instance of derived class.

---

## C++

```

// C++ program for function overriding with data members
#include <bits/stdc++.h>
using namespace std;

#include <iostream>
using namespace std;
class Animal {                                     // base class declaration.
public:
    string color = "Black";
};

class Dog: public Animal                         // inheriting Animal class.
{
public:
    string color = "Grey";
};

//Driver code
int main(void) {
    Animal d= Dog();   //accessing the field by reference variable which refers to der
    cout<<d.color;
}

```



## Output

```
black
```

## Virtual Function

A virtual function is a member function that is declared in the base class using the keyword `virtual` and is re-defined (Overridden) in the derived class.

### Some Key Points About Virtual Functions:

- Virtual functions are Dynamic in nature.
- They are defined by inserting the keyword "**virtual**" inside a base class and are always declared with a base class and overridden in a child class
- A virtual function is called during Runtime

Below is the C++ program to demonstrate virtual function:

## C++

```
// C++ Program to demonstrate
// the Virtual Function
#include <iostream>
using namespace std;

// Declaring a Base class
class GFG_Base {

public:
    // virtual function
    virtual void display()
    {
        cout << "Called virtual Base Class function" <<
            "\n\n";
    }

    void print()
    {
        cout << "Called GFG_Base print function" <<
            "\n\n";
    }
};

// Declaring a Child Class
class GFG_Child : public GFG_Base {

public:
    void display()
    {
        cout << "Called GFG_Child Display Function" <<
            "\n\n";
    }

    void print()
    {
        cout << "Called GFG_Child print Function" <<
            "\n\n";
    }
};
```

```
// Driver code
int main()
{
    // Create a reference of class GFG_Base
    GFG_Base* base;

    GFG_Child child;

    base = &child;

    // This will call the virtual function
    base->GFG_Base::display();

    // this will call the non-virtual function
    base->print();
}
```

## Output

Called virtual Base Class function

Called GFG\_Base print function

This article is contributed by **Harsh Agarwal**. If you like GeeksforGeeks and would like to contribute, you can also write an article using [write.geeksforgeeks.org](https://www.geeksforgeeks.org/write-for-us/) or mail your article to review-team@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks. Please write comments if you find anything incorrect, or if you want to share more information about the topic discussed above.

704

## Related Articles

1. [Virtual Functions and Runtime Polymorphism in C++](#)

---
2. [Difference between Inheritance and Polymorphism](#)

---
3. [Runtime Polymorphism in various types of Inheritance in C++](#)

---
4. [C++ Error – Does not name a type](#)

---
5. [Execution Policy of STL Algorithms in Modern C++](#)

---
6. [C++ Program To Print Pyramid Patterns](#)

---

**SALE!**  
**GeeksforGeeks Courses Upto 25% Off Enroll Now!**



Save 25% on Courses DSA Data Structures Algorithms Interview Preparation Data Science T

# Encapsulation in C++

Difficulty Level : Easy • Last Updated : 18 Feb, 2023

[Read](#) [Discuss](#) [Courses](#) [Practice](#) [Video](#)

Encapsulation in C++ is defined as the wrapping up of data and information in a single unit. In Object Oriented Programming, Encapsulation is defined as binding together the data and the functions that manipulate them.

Consider a real-life example of encapsulation, in a company, there are different sections like the accounts section, finance section, sales section, etc. Now,

- The finance section handles all the financial transactions and keeps records of all the data related to finance.
- Similarly, the sales section handles all the sales-related activities and keeps records of all the sales.

Now there may arise a situation when for some reason an official from the finance section needs all the data about sales in a particular month.

In this case, he is not allowed to directly access the data of the sales section. He will first have to contact some other officer in the sales section and then request him to give the particular data.

AD

This is what **Encapsulation** is. Here the data of the sales section and the employees that can manipulate them are wrapped under a single name "sales section".

## Features of Encapsulation

Below are the features of encapsulation:

1. We can not access any function from the class directly. We need an object to access that function that is using the member variables of that class.
2. The function which we are making inside the class must use only member variables, only then it is called *encapsulation*.
3. If we don't make a function inside the class which is using the member variable of the class then we don't call it encapsulation.
4. Increase in the security of data
5. It helps to control the modification of our data members.

## Encapsulation in C++



## Class

Encapsulation also leads to [data abstraction](#). Using encapsulation also hides the data, as in the above example, the data of the sections like sales, finance, or accounts are hidden from any other section.

### Simple Example of C++:

## C++

```
#include <iostream>
#include <string>

using namespace std;

class Person {
private:
    string name;
    int age;
public:
    Person(string name, int age) {
        this->name = name;
        this->age = age;
    }
    void setName(string name) {
        this->name = name;
    }
    string getName() {
        return name;
    }
    void setAge(int age) {
        this->age = age;
    }
    int getAge() {
        return age;
    }
};

int main() {
    Person person("John Doe", 30);

    cout << "Name: " << person.getName() << endl;
    cout << "Age: " << person.getAge() << endl;

    person.setName("Jane Doe");
    person.setAge(32);

    cout << "Name: " << person.getName() << endl;
    cout << "Age: " << person.getAge() << endl;

    return 0;
}
```

Output:

Name: John Doe

Age: 30

Name: Jane Doe

Age: 32

In C++, encapsulation can be implemented using **classes** and [access modifiers](#).

### Example:

---

## C++

```
// C++ program to demonstrate
// Encapsulation
#include <iostream>
using namespace std;

class Encapsulation {
private:
    // Data hidden from outside world
    int x;

public:
    // Function to set value of
    // variable x
    void set(int a) { x = a; }

    // Function to return value of
    // variable x
    int get() { return x; }
};

// Driver code
int main()
{
    Encapsulation obj;
    obj.set(5);
    cout << obj.get();
    return 0;
}
```

## Output

5

**Explanation:** In the above program, the variable **x** is made private. This variable can be accessed and manipulated only using the functions **get()** and **set()** which are present inside the class. Thus we can say that here, the variable **x** and the functions **get()** and **set()** are bound together which is nothing but encapsulation.

---

## C++

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

// declaring class
```

```

class Circle {
    // access modifier
private:
    // Data Member
    float area;
    float radius;

public:
    void getRadius()
    {
        cout << "Enter radius\n";
        cin >> radius;
    }
    void findArea()
    {
        area = 3.14 * radius * radius;
        cout << "Area of circle=" << area;
    }
};

int main()
{
    // creating instance(object) of class
    Circle cir;
    cir.getRadius(); // calling function
    cir.findArea(); // calling function
}

```

## Output

Enter radius  
Area of circle=0

## Role of Access Specifiers in Encapsulation

Access specifiers facilitate Data Hiding in C++ programs by restricting access to the class member functions and data members. There are three types of access specifiers in C++:

- **Private**
- **Protected**
- **Public**

By **default**, all data members and member functions of a class are made **private** by the compiler.

## Points to Consider

As we have seen in the above example, access specifiers play an important role in implementing encapsulation in C++. The process of implementing encapsulation can be sub-divided into two steps:

1. Creating a class to encapsulate all the data and methods into a single unit.
2. Hiding relevant data using access specifiers.

This article is contributed by **Harsh Agarwal**. If you like GeeksforGeeks and would like to contribute, you can also write an article using [write.geeksforgeeks.org](https://write.geeksforgeeks.org). See your article appearing on the GeeksforGeeks main page and help other Geeks. Please write comments if you find anything incorrect, or if you want to share more information about the topic discussed above.

478

## Related Articles

1. [Difference between Abstraction and Encapsulation in C++](#)

---

2. [C++ Error - Does not name a type](#)

---

3. [Execution Policy of STL Algorithms in Modern C++](#)

---

4. [C++ Program To Print Pyramid Patterns](#)

---

5. [Jagged Arrays in C++](#)

---

6. [Introduction to Parallel Programming with OpenMP in C++](#)

---

7. [Hollow Half Pyramid Pattern Using Numbers](#)

---

8. [30 OOPs Interview Questions and Answers \(2023\)](#)

---

9. [C++ <cstdio>](#)

---

10. [C++ <cstring>](#)

---

Previous

Next

## Article Contributed By :



GeeksforGeeks

SALE!  
GeeksforGeeks Courses Upto 25% Off Enroll Now!



Save 25% on Courses DSA Data Structures Algorithms Interview Preparation Data Science T

# Abstraction in C++

Difficulty Level : Easy • Last Updated : 23 Dec, 2022

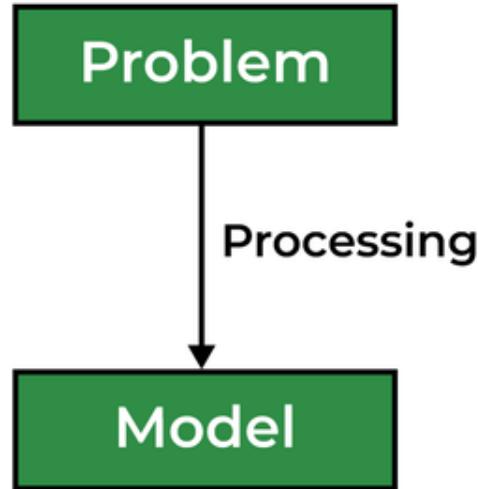
[Read](#) [Discuss](#) [Courses](#) [Practice](#) [Video](#)

Data abstraction is one of the most essential and important features of object-oriented programming in C++. Abstraction means displaying only essential information and hiding the details. Data abstraction refers to providing only essential information about the data to the outside world, hiding the background details or implementation.

Consider a ***real-life example of a man driving a car***. The man only knows that pressing the accelerator will increase the speed of the car or applying brakes will stop the car but he does not know how on pressing the accelerator the speed is actually increasing, he does not know about the inner mechanism of the car or the implementation of the accelerator, brakes, etc in the car. This is what abstraction is.

## Types of Abstraction:

1. **Data abstraction** – This type only shows the required information about the data and hides the unnecessary data.
2. **Control Abstraction** – This type only shows the required information about the implementation and hides unnecessary information.



## Abstraction using Classes

We can implement Abstraction in C++ using classes. The class helps us to group data members and member functions using available access specifiers. A Class can decide which data member will be visible to the outside world and which is not.

## Abstraction in Header files

One more type of abstraction in C++ can be header files. For example, consider the `pow()` method present in `math.h` header file. Whenever we need to calculate the power of a number, we simply call the function `pow()` present in the `math.h` header file and pass the numbers as arguments without knowing the underlying algorithm according to which the function is actually calculating the power of numbers.

AD

## Abstraction using Access Specifiers

Access specifiers are the main pillar of implementing abstraction in C++. We can use access specifiers to enforce restrictions on class members. For example:

- Members declared as **public** in a class can be accessed from anywhere in the program.
- Members declared as **private** in a class, can be accessed only from within the class.  
They are not allowed to be accessed from any part of the code outside the class.

We can easily implement abstraction using the above two features provided by access specifiers. Say, the members that define the internal implementation can be marked as private in a class. And the important information needed to be given to the outside world can be marked as public. And these public members can access the private members as they are inside the class.

### Example:

---

## C++

```
// C++ Program to Demonstrate the
// working of Abstraction
#include <iostream>
using namespace std;

class implementAbstraction {
private:
    int a, b;

public:
    // method to set values of
    // private members
    void set(int x, int y)
    {
        a = x;
        b = y;
    }

    void display()
    {
        cout << "a = " << a << endl;
        cout << "b = " << b << endl;
    }
};

int main()
{
    implementAbstraction obj;
    obj.set(10, 20);
    obj.display();
    return 0;
}
```

## Output

```
a = 10
b = 20
```

You can see in the above program we are not allowed to access the variables a and b directly, however, one can call the function set() to set the values in a and b and the

function display() to display the values of a and b.

## Advantages of Data Abstraction

- Helps the user to avoid writing the low-level code
- Avoids code duplication and increases reusability.
- Can change the internal implementation of the class independently without affecting the user.
- Helps to increase the security of an application or program as only important details are provided to the user.
- It reduces the complexity as well as the redundancy of the code, therefore increasing the readability.

This article is contributed by **Harsh Agarwal**. If you like GeeksforGeeks and would like to contribute, you can also write an article using [write.geeksforgeeks.org](https://www.geeksforgeeks.org/write-for-us/) or mail your article to review-team@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks. Please write comments if you find anything incorrect, or if you want to share more information about the topic discussed above.

## C++

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

class Vehicle
{
public:
    void company()
    {
        cout<<"GFG\n";
    }
public:
    void model()
    {
        cout<<"SIMPLE\n";
    }
public:
    void color()
    {
        cout<<"Red/GREEN/Silver\n";
    }
public:
    void cost()
    {
        cout<<"Rs. 60000 to 900000\n";
    }
public:
    void oil()
    {
        cout<<"PETRO\n";
    }
}
```

```
        }
private:
    void piston()
    {
        cout<<"4 piston\n";
    }
private:
    void manWhoMade()
    {
        cout<<"Markus Librette\n";
    }
};

int main()
{
    Vehicle obj;
    obj.company();
    obj.model();
    obj.color();
    obj.cost();
    obj.oil();
}
```

## Output

GFG  
SIMPLE  
Red/GREEN/Silver  
Rs. 60000 to 900000  
PETRO

435

## Related Articles

1. Difference between Abstraction and Encapsulation in C++

---
2. C++ Error - Does not name a type

---
3. Execution Policy of STL Algorithms in Modern C++

---
4. C++ Program To Print Pyramid Patterns

---
5. Jagged Arrays in C++

---
6. Introduction to Parallel Programming with OpenMP in C++

---

**SALE!**  
**GeeksforGeeks Courses Upto 25% Off Enroll Now!**

[Save 25% on Courses](#)[DSA](#)[Data Structures](#)[Algorithms](#)[Interview Preparation](#)[Data Science](#)[T](#)

# Function Overloading in C++

Difficulty Level : Easy • Last Updated : 16 Mar, 2023

[Read](#) [Discuss](#) [Courses](#) [Practice](#) [Video](#)

Function overloading is a feature of object-oriented programming where two or more functions can have the same name but different parameters. When a function name is overloaded with different jobs it is called Function Overloading. In Function Overloading "Function" name should be the same and the arguments should be different. Function overloading can be considered as an example of a [polymorphism](#) feature in C++.

If multiple functions having same name but parameters of the functions should be different is known as Function Overloading.

If we have to perform only one operation and having same name of the functions increases the readability of the program.

Suppose you have to perform addition of the given numbers but there can be any number of arguments, if you write the function such as `a(int,int)` for two parameters, and `b(int,int,int)` for three parameters then it may be difficult for you to understand the behavior of the function because its name differs.

The parameters should follow any one or more than one of the following conditions for Function overloading:

- Parameters should have a different type

`add(int a, int b)`

`add(double a, double b)`

AD

Below is the implementation of the above discussion:

---

## C++

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

void add(int a, int b)
{
    cout << "sum = " << (a + b);
}

void add(double a, double b)
{
    cout << endl << "sum = " << (a + b);
}

// Driver code
int main()
{
    add(10, 2);
    add(5.3, 6.2);

    return 0;
}
```

## Output

```
sum = 12
sum = 11.5
```

- Parameters should have a different number

*add(int a, int b)*

*add(int a, int b, int c)*

Below is the implementation of the above discussion:

## C++

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

void add(int a, int b)
{
    cout << "sum = " << (a + b);
}

void add(int a, int b, int c)
{
    cout << endl << "sum = " << (a + b + c);
}

// Driver code
int main()
{
    add(10, 2);
    add(5, 6, 4);

    return 0;
}
```

## Output

```
sum = 12
sum = 15
```

- Parameters should have a different sequence of parameters.

*add(int a, double b)  
add(double a, int b)*

Below is the implementation of the above discussion:

## C++

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

```

void add(int a, double b)
{
    cout<<"sum = "<<(a+b);
}

void add(double a, int b)
{
    cout<<endl<<"sum = "<<(a+b);
}

// Driver code
int main()
{
    add(10,2.5);
    add(5.5,6);

    return 0;
}

```

## Output

sum = 12.5  
 sum = 11.5

Following is a simple C++ example to demonstrate function overloading.

---

## CPP

```

#include <iostream>
using namespace std;

void print(int i) {
    cout << " Here is int " << i << endl;
}
void print(double f) {
    cout << " Here is float " << f << endl;
}
void print(char const *c) {
    cout << " Here is char* " << c << endl;
}

int main() {
    print(10);
    print(10.10);
    print("ten");
    return 0;
}

```

## Output

```
Here is int 10
Here is float 10.1
Here is char* ten
```

---

## C++

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

void add(int a, int b)
{
    cout<<"sum ="<<(a+b);
}

void add(int a, int b,int c)
{
    cout<<endl<<"sum ="<<(a+b+c);
}

main()
{
    add(10,2);
    add(5,6,4);
    return 0;
}
```

---

## C++

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

void add(int a, double b)
{
    cout<<"sum ="<<(a+b);
}
void add(double a, int b)
{
    cout<<endl<<"sum ="<<(a+b);
}

main()
{
    add(10,2.5);
    add(5.5,6);
    return 0;
}
```

## How does Function Overloading work?

- *Exact match:-* (Function name and Parameter)
- *If a not exact match is found:-*

->Char, Unsigned char, and short are promoted to an int.

->Float is promoted to double

- *If no match is found:*

->C++ tries to find a match through the standard conversion.

- *ELSE ERROR* 😞

1. [Function overloading and return type](#)
2. [Functions that cannot be overloaded in C++](#)
3. [Function overloading and const keyword](#)
4. [Function Overloading vs Function Overriding in C++](#)

### Recent articles on function overloading in C++

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

325

## Related Articles

1. [error: call of overloaded ‘function\(x\)’ is ambiguous | Ambiguity in Function overloading in C++](#)
2. [Function Overloading vs Function Overriding in C++](#)
3. [Function Overloading and Return Type in C++](#)
4. [Function overloading and const keyword](#)
5. [Overloading function templates in C++](#)
6. [Advantages and Disadvantages of Function Overloading in C++](#)
7. [Overloading of function-call operator in C++](#)

SALE!  
GeeksforGeeks Courses Upto 25% Off Enroll Now!



Save 25% on Courses DSA Data Structures Algorithms Interview Preparation Data Science T

# Operator Overloading in C++

Difficulty Level : Medium • Last Updated : 05 Mar, 2023

[Read](#) [Discuss\(30+\)](#) [Courses](#) [Practice](#) [Video](#)

***Operator overloading is a compile-time polymorphism.*** It is an idea of giving special meaning to an existing operator in C++ without changing its original meaning.

In C++, we can make operators work for user-defined classes. This means C++ has the ability to provide the operators with a special meaning for a data type, this ability is known as operator overloading. For example, we can overload an operator '+' in a class like String so that we can concatenate two strings by just using +. Other example classes where arithmetic operators may be overloaded are Complex Numbers, Fractional Numbers, Big integers, etc.

## Example:

```
int a;  
float b, sum;  
sum = a + b;
```

Here, variables "a" and "b" are of types "int" and "float", which are built-in data types. Hence the addition operator '+' can easily add the contents of "a" and "b". This is because the addition operator "+" is predefined to add variables of built-in data type only.

AD

**Example:****C++**

```
// C++ Program to Demonstrate the
// working/Logic behind Operator
// Overloading
class A {
    statements;
};

int main()
{
    A a1, a2, a3;

    a3 = a1 + a2;

    return 0;
}
```

In this example, we have 3 variables "a1", "a2" and "a3" of type "class A". Here we are trying to add two objects "a1" and "a2", which are of user-defined type i.e. of type "class A" using the "+" operator. This is not allowed, because the addition operator "+" is predefined to operate only on built-in data types. But here, "class A" is a user-defined type, so the compiler generates an error. This is where the concept of "Operator overloading" comes in.

Now, if the user wants to make the operator "+" add two class objects, the user has to redefine the meaning of the "+" operator such that it adds two class objects. This is done by using the concept of "Operator overloading". So the main idea behind "Operator overloading" is to use C++ operators with class variables or class objects. Redefining the meaning of operators really does not change their original meaning; instead, they have been given additional meaning along with their existing ones.

**Example:****C++**

```
// C++ Program to Demonstrate
// Operator Overloading
#include <iostream>
using namespace std;

class Complex {
private:
    int real, imag;

public:
    Complex(int r = 0, int i = 0)
```

```

{
    real = r;
    imag = i;
}

// This is automatically called when '+' is used with
// between two Complex objects
Complex operator+(Complex const& obj)
{
    Complex res;
    res.real = real + obj.real;
    res.imag = imag + obj.imag;
    return res;
}
void print() { cout << real << " + i" << imag << '\n'; }

};

int main()
{
    Complex c1(10, 5), c2(2, 4);
    Complex c3 = c1 + c2;
    c3.print();
}

```

## Output

12 + i9

## What is the difference between operator functions and normal functions?

Operator functions are the same as normal functions. The only differences are, that the name of an operator function is always the operator keyword followed by the symbol of the operator, and operator functions are called when the corresponding operator is used.

### Example:

---

## C++

```

#include <iostream>
using namespace std;
class Complex {
private:
    int real, imag;

public:
    Complex(int r = 0, int i = 0)
    {
        real = r;
        imag = i;
    }
    void print() { cout << real << " + i" << imag << endl; }
}

```

```

// The global operator function is made friend of this
// class so that it can access private members
friend Complex operator+(Complex const& c1,
                         Complex const& c2);

};

Complex operator+(Complex const& c1, Complex const& c2)
{
    return Complex(c1.real + c2.real, c1.imag + c2.imag);
}

int main()
{
    Complex c1(10, 5), c2(2, 4);
    Complex c3
        = c1
        + c2; // An example call to "operator+"
    c3.print();
    return 0;
}

```

## Output

12 + i9

## Can we overload all operators?

Almost all operators can be overloaded except a few. Following is the list of operators that cannot be overloaded.

- sizeof
- typeid
- Scope resolution (::)
- Class member access operators (.(dot), .\* (pointer to member operator))
- Ternary or conditional (?:)

## Operators that can be overloaded

We can overload

- **Unary operators**
- **Binary operators**
- **Special operators** ( [ ], ( ), etc)

But, among them, there are some operators that cannot be overloaded. They are

- **Scope resolution operator (:** 😊)
- **Member selection operator**
- **Member selection through \***

## Pointer to a member variable

- **Conditional operator (?)**
- **Sizeof operator sizeof()**

Operators that can be overloaded	Examples
Binary Arithmetic	+ , - , * , / , %
Unary Arithmetic	+ , - , ++ , -
Assignment	= , += , *= , /= , -= , %=
Bitwise	& ,   , << , >> , ~ , ^
De-referencing	(->)
Dynamic memory allocation, De-allocation	New, delete
Subscript	[ ]
Function call	()
Logical	& ,    , !
Relational	> , < , == , <= , >=

## Why can't the above-stated operators be overloaded?

**1. sizeof** – This returns the size of the object or datatype entered as the operand. This is evaluated by the compiler and cannot be evaluated during runtime. The proper incrementing of a pointer in an array of objects relies on the sizeof operator implicitly. Altering its meaning using overloading would cause a fundamental part of the language to collapse.

**2. typeid**: This provides a CPP program with the ability to recover the actually derived type of the object referred to by a pointer or reference. For this operator, the whole point is to uniquely identify a type. If we want to make a user-defined type 'look' like another type,

polymorphism can be used but the meaning of the typeid operator must remain unaltered, or else serious issues could arise.

**3. Scope resolution (::):** This helps identify and specify the context to which an identifier refers by specifying a namespace. It is completely evaluated at runtime and works on names rather than values. The operands of scope resolution are note expressions with data types and CPP has no syntax for capturing them if it were overloaded. So it is syntactically impossible to overload this operator.

**4. Class member access operators (.(dot), .\* (pointer to member operator)):** The importance and implicit use of class member access operators can be understood through the following example:

#### Example:

---

### C++

```
// C++ program to demonstrate operator overloading
// using dot operator
#include <iostream>
using namespace std;

class ComplexNumber {
private:
    int real;
    int imaginary;

public:
    ComplexNumber(int real, int imaginary)
    {
        this->real = real;
        this->imaginary = imaginary;
    }
    void print() { cout << real << " + i" << imaginary; }
    ComplexNumber operator+(ComplexNumber c2)
    {
        ComplexNumber c3(0, 0);
        c3.real = this->real + c2.real;
        c3.imaginary = this->imaginary + c2.imaginary;
        return c3;
    }
};

int main()
{
    ComplexNumber c1(3, 5);
    ComplexNumber c2(2, 4);
    ComplexNumber c3 = c1 + c2;
    c3.print();
    return 0;
}
```

## Output

```
5 + i9
```

### Explanation:

The statement `ComplexNumber c3 = c1 + c2;` is internally translated as `ComplexNumber c3 = c1.operator+ (c2);` in order to invoke the operator function. The argument `c1` is implicitly passed using the `'.'` operator. The next statement also makes use of the dot operator to access the member function `print` and pass `c3` as an argument.

Besides, these operators also work on names and not values and there is no provision (syntactically) to overload them.

**5. Ternary or conditional (?:):** The ternary or conditional operator is a shorthand representation of an if-else statement. In the operator, the true/false expressions are only evaluated on the basis of the truth value of the conditional expression.

```
conditional statement ? expression1 (if statement is TRUE) : expression2  
(else)
```

A function overloading the ternary operator for a class say ABC using the definition

```
ABC operator ?: (bool condition, ABC trueExpr, ABC falseExpr);
```

would not be able to guarantee that only one of the expressions was evaluated. Thus, the ternary operator cannot be overloaded.

## Important points about operator overloading

**1)** For operator overloading to work, at least one of the operands must be a user-defined class object.

**2) Assignment Operator:** Compiler automatically creates a default assignment operator with every class. The default assignment operator does assign all members of the right side to the left side and works fine in most cases (this behavior is the same as the copy constructor). See [this](#) for more details.

**3) Conversion Operator:** We can also write conversion operators that can be used to convert one type to another type.

### Example:

---

## C++

```
// C++ Program to Demonstrate the working
```

```
// of conversion operator
#include <iostream>
using namespace std;
class Fraction {
private:
    int num, den;

public:
    Fraction(int n, int d)
    {
        num = n;
        den = d;
    }

    // Conversion operator: return float value of fraction
    operator float() const
    {
        return float(num) / float(den);
    }
};

int main()
{
    Fraction f(2, 5);
    float val = f;
    cout << val << '\n';
    return 0;
}
```

## Output

0.4

Overloaded conversion operators must be a member method. Other operators can either be the member method or the global method.

**4)** Any constructor that can be called with a single argument works as a conversion constructor, which means it can also be used for implicit conversion to the class being constructed.

### Example:

---

## C++

```
// C++ program to demonstrate can also be used for implicit
// conversion to the class being constructed
#include <iostream>
using namespace std;

class Point {
private:
    int x, y;
```

```

public:
    Point(int i = 0, int j = 0)
    {
        x = i;
        y = j;
    }
    void print()
    {
        cout << "x = " << x << ", y = " << y << '\n';
    }
};

int main()
{
    Point t(20, 20);
    t.print();
    t = 30; // Member x of t becomes 30
    t.print();
    return 0;
}

```

## Output

x = 20, y = 20  
x = 30, y = 0

### Quiz on Operator Overloading

356

## Related Articles

1. Operator Overloading '<<' and '>>' operator in a linked list class

---

2. Rules for operator overloading

---

3. Overloading Subscript or array index operator [] in C++

---

4. C++ | Operator Overloading | Question 10

---

5. Overloading New and Delete operator in c++

---

6. C++ Program to concatenate two strings using Operator Overloading

**SALE!**  
**GeeksforGeeks Courses Upto 25% Off Enroll Now!**



Save 25% on Courses DSA Data Structures Algorithms Interview Preparation Data Science T

# Constructors in C++

Difficulty Level : Easy • Last Updated : 27 Mar, 2023

[Read](#) [Discuss\(20+\)](#) [Courses](#) [Practice](#) [Video](#)

**Constructor in C++** is a special method that is invoked automatically at the time of object creation. It is used to initialize the data members of new objects generally. The constructor in C++ has the same name as the class or structure. Constructor is invoked at the time of object creation. It constructs the values i.e. provides data for the object which is why it is known as constructors.

- Constructor is a member function of a class, whose name is same as the class name.
- Constructor is a special type of member function that is used to initialize the data members for an object of a class automatically, when an object of the same class is created.
- Constructor is invoked at the time of object creation. It constructs the values i.e. provides data for the object that is why it is known as constructor.
- Constructor do not return value, hence they do not have a return type.

The prototype of the constructor looks like

```
<class-name> (list-of-parameters);
```

Constructor can be defined inside the class declaration or outside the class declaration

a. Syntax for defining the constructor within the class

```
<class-name>(list-of-parameters)
{
    //constructor definition
```

```
}
```

- b. Syntax for defining the constructor outside the class

```
<class-name> : <class-name>(list-of-parameters)
{
    //constructor definition
}
```

AD

---

## C++

```
// Example: defining the constructor within the class
```

```
#include<iostream>
using namespace std;
class student
{
    int rno;
    char name[50];
    double fee;
public:
student()
{
    cout<<"Enter the RollNo:";
    cin>>rno;
    cout<<"Enter the Name:";
    cin>>name;
    cout<<"Enter the Fee:";
    cin>>fee;
}
void display()
{
    cout<<endl<<rno<<"\t"<<name<<"\t"<<fee;
}
};
```

```
int main()
{
    student s; //constructor gets called automatically when we create the object of t
    s.display();
    return 0;
}
```

## C++

```
// Example: defining the constructor outside the class

#include<iostream>
using namespace std;
class student
{
    int rno;
    char name[50];
    double fee;
public:
    student();
    void display();

};

student::student()
{
    cout<<"Enter the RollNo:";
    cin>>rno;
    cout<<"Enter the Name:";
    cin>>name;
    cout<<"Enter the Fee:";
    cin>>fee;
}

void student::display()
{
    cout<<endl<<rno<<"\t"<<name<<"\t"<<fee;
}

int main()
{
    student s;
    s.display();
    return 0;
}
```

## Characteristics of constructor

- The name of the constructor is same as its class name.
- Constructors are mostly declared in the public section of the class though it can be declared in the private section of the class.
- Constructors do not return values; hence they do not have a return type.
- A constructor gets called automatically when we create the object of the class.
- Constructors can be overloaded.
- Constructor can not be declared virtual.

## Types of constructor

- Default constructor
- Parameterized constructor
- Overloaded constructor
- Constructor with default value
- Copy constructor
- Inline constructor

Constructor does not have a return value, hence they do not have a return type.

The prototype of Constructors is as follows:

```
<class-name> (list-of-parameters);
```

Constructors can be defined inside or outside the class declaration:-

The syntax for defining the constructor within the class:

```
<class-name> (list-of-parameters) { // constructor definition }
```

The syntax for defining the constructor outside the class:

```
<class-name> : <class-name> (list-of-parameters){ // constructor definition}
```

## Example

---

### C++

```
// defining the constructor within the class

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

```

class student {
    int rno;
    char name[10];
    double fee;

public:
    student()
    {
        cout << "Enter the RollNo:";
        cin >> rno;
        cout << "Enter the Name:";
        cin >> name;
        cout << "Enter the Fee:";
        cin >> fee;
    }

    void display()
    {
        cout << endl << rno << "\t" << name << "\t" << fee;
    }
};

int main()
{
    student s; // constructor gets called automatically when
               // we create the object of the class
    s.display();

    return 0;
}

```

## Output

Enter the RollNo:Enter the Name:Enter the Fee:  
0 6.95303e-310

## Example

---

### C++

```

// defining the constructor outside the class

#include <iostream>
using namespace std;
class student {
    int rno;
    char name[50];
    double fee;

public:
    student();
    void display();

```

```

};

student::student()
{
    cout << "Enter the RollNo:";
    cin >> rno;

    cout << "Enter the Name:";
    cin >> name;

    cout << "Enter the Fee:";
    cin >> fee;
}

void student::display()
{
    cout << endl << rno << "\t" << name << "\t" << fee;
}

int main()
{
    student s;
    s.display();

    return 0;
}

```

## Output:

```

Enter the RollNo: 30
Enter the Name: ram
Enter the Fee: 20000
30 ram 20000

```

## How constructors are different from a normal member function?

---

## C++

```

#include <iostream>
using namespace std;

class Line {
public:
    void setLength( double len );
    double getLength( void );
    Line( double len ); //This is the constructor
private:
    double length;
};
//Member function definition including constructor
Line::Line( double len ) {

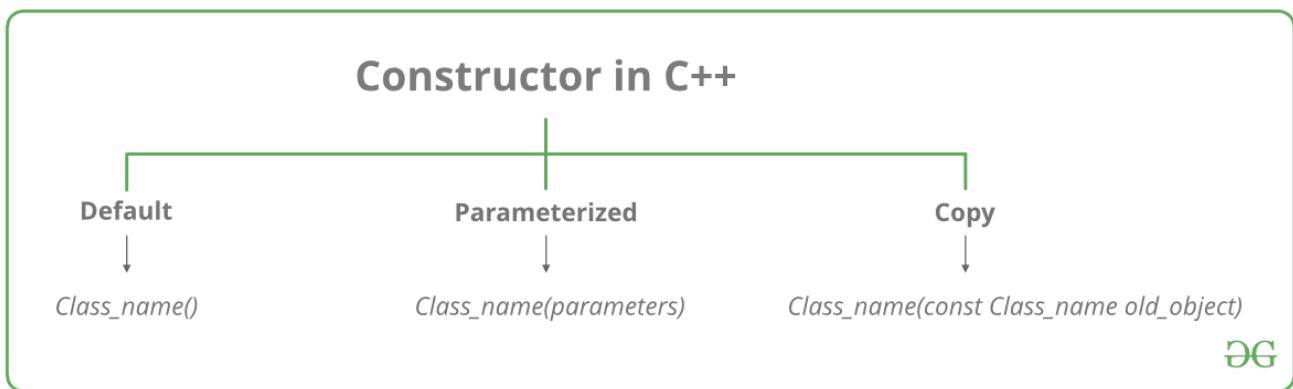
```

```
cout<<"Object is being created , length ="<< len << endl;
length = len;
}
void Line::setLength( double len ) {
    length = len;
}
double Line::getLength( void ) {
    return length;
}
//Main function for the program
int main() {
    Line line(10.0);
    //get initially set length
    cout<<"Length of line :" << line.getLength() << endl;
    //set line length again
    line.setLength(6.0);
    cout<<"Length of line :" << line.getLength() << endl;

    return 0;
}
```

A constructor is different from normal functions in following ways:

- Constructor has same name as the class itself
- Default Constructors don't have input argument however, Copy and Parameterized Constructors have input arguments
- Constructors don't have return type
- A constructor is automatically called when an object is created.
- It must be placed in public section of class.
- If we do not specify a constructor, C++ compiler generates a default constructor for object (expects no parameters and has an empty body).



Let us understand the types of constructors in C++ by taking a real-world example.

Suppose you went to a shop to buy a marker. When you want to buy a marker, what are the options. The first one you go to a shop and say give me a marker. So just saying give me a marker mean that you did not set which brand name and which color, you didn't mention anything just say you want a marker. So when we said just I want a marker so whatever the frequently sold marker is there in the market or in his shop he will simply hand over that. And this is what a default constructor is! The second method is you go to a shop and say I want a marker a red in color and XYZ brand. So you are mentioning this and he will give you that marker. So in this case you have given the parameters. And this is what a parameterized constructor is! Then the third one you go to a shop and say I want a marker like this(a physical marker on your hand). So the shopkeeper will see that marker. Okay, and he will give a new marker for you. So copy of that marker. And that's what a copy constructor is!

### Characteristics of the constructor:

- The name of the constructor is the same as its class name.
- Constructors are mostly declared in the public section of the class though it can be declared in the private section of the class.
- Constructors do not return values; hence they do not have a return type.
- A constructor gets called automatically when we create the object of the class.
- Constructors can be overloaded.
- Constructor can not be declared virtual.
- Constructor cannot be inherited.
- Addresses of Constructor cannot be referred.
- Constructor make implicit calls to **new** and **delete** operators during memory allocation.

### Types of Constructors

**1. Default Constructors:** Default constructor is the constructor which doesn't take any argument. It has no parameters. It is also called a zero-argument constructor.

---

## CPP

```
// Cpp program to illustrate the
// concept of Constructors
#include <iostream>
using namespace std;

class construct {
public:
    int a, b;

    // Default Constructor
    construct()
    {
        a = 10;
        b = 20;
    }
};

int main()
{
    // Default constructor called automatically
    // when the object is created
    construct c;
    cout << "a: " << c.a << endl << "b: " << c.b;
    return 1;
}
```

## Output

```
a: 10
b: 20
```

**Note:** Even if we do not define any constructor explicitly, the compiler will automatically provide a default constructor implicitly.

---

## C++

```
// Example
#include<iostream>
using namespace std;
class student
```

```

{
    int rno;
    char name[50];
    double fee;
    public:
    student() // Explicit Default constructor
    {
        cout<<"Enter the RollNo:";
        cin>>rno;
        cout<<"Enter the Name:";
        cin>>name;
        cout<<"Enter the Fee:";
        cin>>fee;
    }

    void display()
    {
        cout<<endl<<rno<<"\t"<<name<<"\t"<<fee;
    }
};

int main()
{
    student s;
    s.display();
    return 0;
}

```

**2. Parameterized Constructors:** It is possible to pass arguments to constructors. Typically, these arguments help initialize an object when it is created. To create a parameterized constructor, simply add parameters to it the way you would to any other function. When you define the constructor's body, use the parameters to initialize the object.

*Note: when the parameterized constructor is defined and no default constructor is defined explicitly, the compiler will not implicitly call the default constructor and hence creating a simple object as*

*Student s;  
Will flash an error*

## CPP

```

// CPP program to illustrate
// parameterized constructors
#include <iostream>
using namespace std;

```

```

class Point {
private:
    int x, y;

public:
    // Parameterized Constructor
    Point(int x1, int y1)
    {
        x = x1;
        y = y1;
    }

    int getX() { return x; }
    int getY() { return y; }
};

int main()
{
    // Constructor called
    Point p1(10, 15);

    // Access values assigned by constructor
    cout << "p1.x = " << p1.getX()
       << ", p1.y = " << p1.getY();

    return 0;
}

```

## Output

p1.x = 10, p1.y = 15

---

## C++

```

// Example

#include<iostream>
#include<string.h>
using namespace std;

class student
{
    int rno;
    char name[50];
    double fee;

    public:
    student(int,char["],double);
    void display();

};

```

```

student::student(int no,char n[],double f)
{
    rno=no;
    strcpy(name,n);
    fee=f;
}

void student::display()
{
    cout<<endl<<rno<<"\t"<<name<<"\t"<<fee;
}

int main()
{
    student s(1001,"Ram",10000);
    s.display();
    return 0;
}

```

When an object is declared in a parameterized constructor, the initial values have to be passed as arguments to the constructor function. The normal way of object declaration may not work. The constructors can be called explicitly or implicitly.

Example e = Example(0, 50); // Explicit call

Example e(0, 50); // Implicit call

- **Uses of Parameterized constructor:**

1. It is used to initialize the various data elements of different objects with different values when they are created.
2. It is used to overload constructors.

- **Can we have more than one constructor in a class?**

Yes, It is called [Constructor Overloading](#).

### 3. Copy Constructor:

A copy constructor is a member function that initializes an object using another object of the same class. A detailed article on [Copy Constructor](#).

Whenever we define one or more non-default constructors (with parameters) for a class, a default constructor (without parameters) should also be explicitly defined as the compiler will not provide a default constructor in this case. However, it is not necessary but it's considered to be the best practice to always define a default constructor.

Copy constructor takes a reference to an object of the same class as an argument.

```
Sample(Sample &t)
{
    id=t.id;
}
```

---

## CPP

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

class point {
private:
    double x, y;

public:
    // Non-default Constructor &
    // default Constructor
    point(double px, double py) { x = px, y = py; }
};

int main(void)
{
    // Define an array of size
    // 10 & of type point
    // This line will cause error
    point a[10];

    // Remove above line and program
    // will compile without error
    point b = point(5, 6);
}
```

## Output:

Error: point (double px, double py): expects 2 arguments, 0 provided

---

## C++

```
// Implicit copy constructor

#include<iostream>
using namespace std;

class Sample
{
    int id;
public:
    void init(int x)
```

```

{
    id=x;
}
void display()
{
    cout<<endl<<"ID="<<id;
}
};

int main()
{
    Sample obj1;
    obj1.init(10);
    obj1.display();

    Sample obj2(obj1); //or obj2=obj1;
    obj2.display();
    return 0;
}

```

## Output

ID=10  
ID=10

---

## C++

```

// Example: Explicit copy constructor

#include <iostream>
using namespace std;

class Sample
{
    int id;
    public:
    void init(int x)
    {
        id=x;
    }
    Sample(){}
        //default constructor with empty body

    Sample(Sample &t)    //copy constructor
    {
        id=t.id;
    }
    void display()
    {
        cout<<endl<<"ID="<<id;
    }
};

int main()
{

```

```

Sample obj1;
obj1.init(10);
obj1.display();

Sample obj2(obj1); //or obj2=obj1;      copy constructor called
obj2.display();
return 0;
}

```

## Output

```

ID=10
ID=10

```

---

## C++

```

#include<iostream>
#include<string.h>
using namespace std;
class student
{
    int rno;
    char name[50];
    double fee;
public:
    student(int,char[],double);
    student(student &t)      //copy constructor
    {
        rno=t.rno;
        strcpy(name,t.name);
        fee=t.fee;
    }
    void display();
};

student::student(int no,char n[],double f)
{
    rno=no;
    strcpy(name,n);
    fee=f;
}

void student::display()
{
    cout<<endl<<rno<<"\t"<<name<<"\t"<<fee;
}

int main()
{

```

```

student s(1001, "Manjeet", 10000);
s.display();

student manjeet(s); //copy constructor called
manjeet.display();

return 0;
}

```

---

## C++

```

#include<iostream>
#include<string.h>
using namespace std;
class student
{
    int rno;
    char name[50];
    double fee;
public:
    student(int, char[], double);
    student(student &t) //copy constructor (member wise initialization)
    {
        rno=t.rno;
        strcpy(name,t.name);

    }
    void display();
    void disp()
    {
        cout<<endl<<rno<<"\t"<<name;
    }
};

student::student(int no, char n[], double f)
{
    rno=no;
    strcpy(name,n);
    fee=f;
}

void student::display()
{
    cout<<endl<<rno<<"\t"<<name<<"\t"<<fee;
}

int main()
{
    student s(1001, "Manjeet", 10000);
    s.display();

    student manjeet(s); //copy constructor called

```

```

manjeet.disp();

    return 0;
}

```

## Destructor:

A destructor is also a special member function as a constructor. Destructor destroys the class objects created by the constructor. Destructor has the same name as their class name preceded by a tilde (~) symbol. It is not possible to define more than one destructor. The destructor is only one way to destroy the object created by the constructor. Hence destructor can-not be overloaded. Destructor neither requires any argument nor returns any value. It is automatically called when the object goes out of scope. Destructors release memory space occupied by the objects created by the constructor. In destructor, objects are destroyed in the reverse of object creation.

The syntax for defining the destructor within the class

```

~ <class-name>()
{
}

```

The syntax for defining the destructor outside the class

```
<class-name> : ~ <class-name>(){}  


```

---

## C++

```

#include <iostream>
using namespace std;

class Test {
public:
    Test() { cout << "\n Constructor executed"; }

    ~Test() { cout << "\n Destructor executed"; }
};

main()
{
    Test t;

    return 0;
}

```

## Output

```
Constructor executed  
Destructor executed
```

---

## C++

```
#include <iostream>  
using namespace std;  
class Test {  
public:  
    Test() { cout << "\n Constructor executed"; }  
  
    ~Test() { cout << "\n Destructor executed"; }  
};  
  
main()  
{  
    Test t, t1, t2, t3;  
    return 0;  
}
```

## Output

```
Constructor executed  
Constructor executed  
Constructor executed  
Constructor executed  
Destructor executed  
Destructor executed  
Destructor executed  
Destructor executed
```

---

## C++

```
#include <iostream>  
using namespace std;  
int count = 0;  
class Test {  
public:  
    Test()  
    {  
        count++;  
        cout << "\n No. of Object created:\t" << count;  
    }  
  
    ~Test()  
    {
```

```

        cout << "\n No. of Object destroyed:\t" << count;
        --count;
    }
};

main()
{
    Test t, t1, t2, t3;
    return 0;
}

```

## Output

```

No. of Object created:      1
No. of Object created:      2
No. of Object created:      3
No. of Object created:      4
No. of Object destroyed:    4
No. of Object destroyed:    3
No. of Object destroyed:    2
No. of Object destroyed:    1

```

## Characteristics of a destructor:-

1. Destructor is invoked automatically by the compiler when its corresponding constructor goes out of scope and releases the memory space that is no longer required by the program.
2. Destructor neither requires any argument nor returns any value therefore it cannot be overloaded.
3. Destructor cannot be declared as static and const;
4. Destructor should be declared in the public section of the program.
5. Destructor is called in the reverse order of its constructor invocation.

*Q: What are the functions that are generated by the compiler by default, if we do not provide them explicitly?*

*Ans: The functions that are generated by the compiler by default if we do not provide them explicitly are:*

- I. Default constructor
- II. Copy constructor
- III. Assignment operator
- IV. Destructor

**SALE!**  
**GeeksforGeeks Courses Upto 25% Off Enroll Now!**



Save 25% on Courses DSA Data Structures Algorithms Interview Preparation Data Science T

# Copy Constructor in C++

Difficulty Level : Medium • Last Updated : 16 Mar, 2023

[Read](#) [Discuss\(30\)](#) [Courses](#) [Practice](#) [Video](#)

**Pre-requisite:** [Constructor in C++](#)

A **copy constructor** is a member function that initializes an object using another object of the same class. In simple terms, a constructor which creates an object by initializing it with an object of the same class, which has been created previously is known as a **copy constructor**.

Copy constructor is used to initialize the members of a newly created object by copying the members of an already existing object.

Copy constructor takes a reference to an object of the same class as an argument.

AD

```
Sample(Sample &t)
{
    id=t.id;
}
```

The process of initializing members of an object through a copy constructor is known as copy initialization.

It is also called member-wise initialization because the copy constructor initializes one object with the existing object, both belonging to the same class on a member by member copy basis.

The copy constructor can be defined explicitly by the programmer. If the programmer does not define the copy constructor, the compiler does it for us.

### Example:



**ClassName (const ClassName &old\_obj);**

*Syntax of Copy Constructor*

## C++

```

#include<iostream>
#include<string.h>
using namespace std;
class student
{
    int rno;
    char name[50];
    double fee;
public:
    student(int,char[],double);
    student(student &t)      //copy constructor
    {
        rno=t.rno;
        strcpy(name,t.name);
        fee=t.fee;
    }
    void display();
};
  
```

```

student::student(int no,char n[],double f)
{
    rno=no;
    strcpy(name,n);
    fee=f;
}

void student::display()
{
    cout<<endl<<rno<<"\t"<<name<<"\t"<<fee;
}

int main()
{
    student s(1001,"Manjeet",10000);
    s.display();

    student manjeet(s); //copy constructor called
    manjeet.display();

    return 0;
}

```

## C++

```

#include<iostream>
#include<string.h>
using namespace std;
class student
{
    int rno;
    char name[50];
    double fee;
public:
    student(int,char[],double);
    student(student &t) //copy constructor (member wise initialization)
    {
        rno=t.rno;
        strcpy(name,t.name);

    }
    void display();
    void disp()
    {
        cout<<endl<<rno<<"\t"<<name;
    }
};

student::student(int no, char n[],double f)

```

```

{
    rno=no;
    strcpy(name,n);
    fee=f;
}

void student::display()
{
    cout<<endl<<rno<<"\t"<<name<<"\t"<<fee;
}

int main()
{
    student s(1001,"Manjeet",10000);
    s.display();

    student manjeet(s); //copy constructor called
    manjeet.disp();

    return 0;
}

```

## Characteristics of Copy Constructor

- 1.** The copy constructor is used to initialize the members of a newly created object by copying the members of an already existing object.
- 2.** Copy constructor takes a reference to an object of the same class as an argument.
  

```

Sample(Sample &t)
{
    id=t.id;
}

```

- 3.** The process of initializing members of an object through a copy constructor is known as ***copy initialization***.
- 4.** It is also called member-wise initialization because the copy constructor initializes one object with the existing object, both belonging to the same class on a member-by-member copy basis.
- 5.** The copy constructor can be defined explicitly by the programmer. If the programmer does not define the copy constructor, the compiler does it for us.

### Example:

---

### C++

```
// C++ program to demonstrate the working
// of a COPY CONSTRUCTOR
#include <iostream>
using namespace std;

class Point {
private:
    int x, y;

public:
    Point(int x1, int y1)
    {
        x = x1;
        y = y1;
    }

    // Copy constructor
    Point(const Point& p1)
    {
        x = p1.x;
        y = p1.y;
    }

    int getX() { return x; }
    int getY() { return y; }
};

int main()
{
    Point p1(10, 15); // Normal constructor is called here
    Point p2 = p1; // Copy constructor is called here

    // Let us access values assigned by constructors
    cout << "p1.x = " << p1.getX()
        << ", p1.y = " << p1.getY();
    cout << "\np2.x = " << p2.getX()
        << ", p2.y = " << p2.getY();
    return 0;
}
```

## Output

p1.x = 10, p1.y = 15  
 p2.x = 10, p2.y = 15

## Types of Copy Constructors

### 1. Default Copy Constructor

An implicitly defined copy constructor will copy the bases and members of an object in the same order that a constructor would initialize the bases and members of the object.

---

## C++

```
// Implicit copy constructor Calling
#include <iostream>
using namespace std;

class Sample {
    int id;

public:
    void init(int x) { id = x; }
    void display() { cout << endl << "ID=" << id; }
};

int main()
{
    Sample obj1;
    obj1.init(10);
    obj1.display();

    // Implicit Copy Constructor Calling
    Sample obj2(obj1); // or obj2=obj1;
    obj2.display();
    return 0;
}
```

## Output

```
ID=10
ID=10
```

## 2. User Defined Copy Constructor

A user-defined copy constructor is generally needed when an object owns pointers or non-shareable references, such as to a file, in which case a destructor and an assignment operator should also be written

---

## C++

```
// Explicitly copy constructor Calling
#include<iostream>
using namespace std;

class Sample
{
```

```

int id;
public:
void init(int x)
{
    id=x;
}
Sample(){} //default constructor with empty body

Sample(Sample &t) //copy constructor
{
    id=t.id;
}
void display()
{
    cout<<endl<<"ID="<<id;
}
};

int main()
{
    Sample obj1;
    obj1.init(10);
    obj1.display();

    Sample obj2(obj1); //or obj2=obj1;      copy constructor called
    obj2.display();
    return 0;
}

```

## Output

ID=10  
ID=10

---

## C++

```

// C++ Programt to demonstrate the student details
#include <iostream>
#include <string.h>
using namespace std;
class student {
    int rno;
    string name;
    double fee;

public:
    student(int, string, double);
    student(student& t) // copy constructor
    {
        rno = t.rno;
        name = t.name;
        fee = t.fee;
    }
    void display();

```

```

};

student::student(int no, string n, double f)
{
    rno = no;
    name = n;
    fee = f;
}
void student::display()
{
    cout << endl << rno << "\t" << name << "\t" << fee;
}
int main()
{
    student s(1001, "Ram", 10000);
    s.display();
    student ram(s); // copy constructor called
    ram.display();
    return 0;
}

```

## Output

```

1001      Ram      10000
1001      Ram      10000

```

## When is the copy constructor called?

In C++, a Copy Constructor may be called in the following cases:

- When an object of the class is returned by value.
- When an object of the class is passed (to a function) by value as an argument.
- When an object is constructed based on another object of the same class.
- When the compiler generates a temporary object.

It is, however, not guaranteed that a copy constructor will be called in all these cases, because the C++ Standard allows the compiler to optimize the copy away in certain cases, one example is the **return value optimization** (sometimes referred to as RVO).

## Copy Elision

In [copy elision](#), the compiler prevents the making of extra copies which results in saving space and better the program complexity(both time and space); Hence making the code more optimized.

### Example:

---

## C++

```
// C++ program to demonstrate
// the working of copy elision
#include <iostream>
using namespace std;

class GFG {
public:
    void print() { cout << " GFG!"; }
};

int main()
{
    GFG G;
    for (int i = 0; i <= 2; i++) {
        G.print();
        cout << "\n";
    }
    return 0;
}
```

## Output

GFG!  
GFG!  
GFG!

Now it is on the compiler to decide what it wants to print, it could either print the above output or it could print case 1 or case 2 below, and this is what ***Return Value Optimization*** is. In simple words, ***RVO*** is a technique that gives the compiler some additional power to terminate the temporary object created which results in changing the observable behavior/characteristics of the final program.

### Case 1:

GFG!  
GFG!

### Case 2:

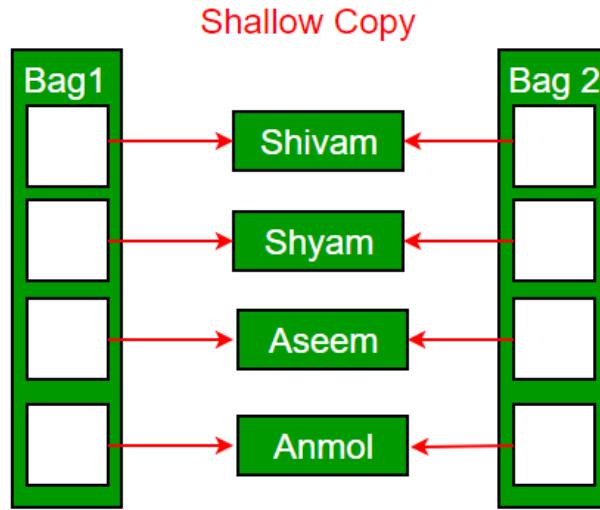
GFG!

## When is a user-defined copy constructor needed?

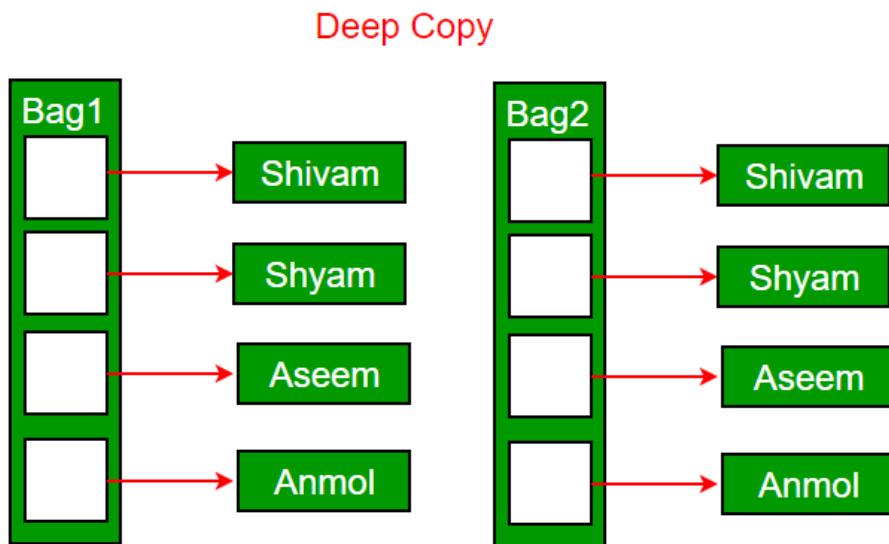
If we don't define our own copy constructor, the C++ compiler creates a default copy constructor for each class which does a member-wise copy between objects. The compiler-created copy constructor works fine in general. We need to define our own copy constructor only if an object has pointers or any runtime allocation of the resource like a file

*handle*, a network connection, etc.

The default **constructor does only shallow copy**.



**Deep copy is possible only with a user-defined copy constructor.** In a user-defined copy constructor, we make sure that pointers (or references) of copied objects point to new memory locations.



## Copy constructor vs Assignment Operator

The main difference between [Copy Constructor and Assignment Operator](#) is that the Copy constructor makes a new memory storage every time it is called while the assignment operator does not make new memory storage.

**Which of the following two statements calls the copy constructor and which one calls the assignment operator?**

```
MyClass t1, t2;
MyClass t3 = t1; // ----> (1)
t2 = t1; // -----> (2)
```

A copy constructor is called when a new object is created from an existing object, as a copy of the existing object. The assignment operator is called when an already initialized object is assigned a new value from another existing object. In the above example (1) calls the copy constructor and (2) calls the assignment operator. See [this](#) for more details.

## Example – Class Where a Copy Constructor is Required

Following is a complete C++ program to demonstrate the use of the Copy constructor. In the following String class, we must write a copy constructor.

### Example:

---

### C++

```
// C++ program to demonstrate the
// Working of Copy constructor
#include <cstring>
#include <iostream>
using namespace std;

class String {
private:
    char* s;
    int size;

public:
    String(const char* str = NULL); // constructor
    ~String() { delete[] s; } // destructor
    String(const String&); // copy constructor
    void print()
    {
        cout << s << endl;
    } // Function to print string
    void change(const char*); // Function to change
};

// In this the pointer returns the CHAR ARRAY
// in the same sequence of string object but
// with an additional null pointer '\0'
String::String(const char* str)
{
    size = strlen(str);
    s = new char[size + 1];
    strcpy(s, str);
}

void String::change(const char* str)
```

```

{
    delete[] s;
    size = strlen(str);
    s = new char[size + 1];
    strcpy(s, str);
}

String::String(const String& old_str)
{
    size = old_str.size;
    s = new char[size + 1];
    strcpy(s, old_str.s);
}

int main()
{
    String str1("GeeksQuiz");
    String str2 = str1;

    str1.print(); // what is printed ?
    str2.print();

    str2.change("GeeksForGeeks");

    str1.print(); // what is printed now ?
    str2.print();
    return 0;
}

```

## Output

```

GeeksQuiz
GeeksQuiz
GeeksQuiz
GeeksforGeeks

```

**What would be the problem if we remove the copy constructor from the above code?**

If we remove the copy constructor from the above program, we don't get the expected output. The changes made to str2 reflect in str1 as well which is never expected.

## C++

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

```

class String {
private:
    char* s;
    int size;

public:
    String(const char* str = NULL); // constructor
    ~String() { delete[] s; } // destructor
    void print() { cout << s << endl; }
    void change(const char*); // Function to change
};

String::String(const char* str)
{
    size = strlen(str);
    s = new char[size + 1];
    strcpy(s, str);
}

// In this the pointer returns the CHAR ARRAY
// in the same sequence of string object but
// with an additional null pointer '\0'
void String::change(const char* str) { strcpy(s, str); }

int main()
{
    String str1("GeeksQuiz");
    String str2 = str1;

    str1.print(); // what is printed ?
    str2.print();

    str2.change("GeeksforGeeks");

    str1.print(); // what is printed now ?
    str2.print();
    return 0;
}

```

## Output:

```

GeeksQuiz
GeeksQuiz
GeeksforGeeks
GeeksforGeeks

```

## Can we make the copy constructor private?

**Yes**, a copy constructor can be made private. When we make a copy constructor private in a class, objects of that class become non-copyable. This is particularly useful when our class has pointers or dynamically allocated resources. In such situations, we can either write our

own copy constructor like the above String example or make a private copy constructor so that users get compiler errors rather than surprises at runtime.

## Why argument to a copy constructor must be passed as a reference?

A copy constructor is called when an object is passed by value. Copy constructor itself is a function. So if we pass an argument by value in a copy constructor, a call to the copy constructor would be made to call the copy constructor which becomes a non-terminating chain of calls. Therefore compiler doesn't allow parameters to be passed by value.

## Why argument to a copy constructor should be const?

One reason for passing ***const*** reference is, that we should use ***const*** in C++ wherever possible so that objects are not accidentally modified. This is one good reason for passing reference as ***const***, but there is more to it than '[Why argument to a copy constructor should be const?](#)'

This article is contributed by **Shubham Agrawal**. If you like GeeksforGeeks and would like to contribute, you can also write your article at [write.geeksforgeeks.org](https://write.geeksforgeeks.org). See your article appearing on the GeeksforGeeks main page and help other Geeks. Please write comments if you find anything incorrect, or if you want to share more information about the topic discussed above.

420

## Related Articles

1. [Copy Constructor vs Assignment Operator in C++](#)

---
2. [When is a Copy Constructor Called in C++?](#)

---
3. [When Should We Write Our Own Copy Constructor in C++?](#)

---
4. [Advanced C++ | Virtual Copy Constructor](#)

---
5. [Why copy constructor argument should be const in C++?](#)

---
6. [Different methods to copy in C++ STL | std::copy\(\), copy\\_n\(\), copy\\_if\(\), copy\\_backward\(\)](#)

---
7. [Shallow Copy and Deep Copy in C++](#)

---

**SALE!**  
**GeeksforGeeks Courses Upto 25% Off Enroll Now!**



Save 25% on Courses DSA Data Structures Algorithms Interview Preparation Data Science T

# Destructors in C++

Difficulty Level : Easy • Last Updated : 11 Dec, 2022

[Read](#) [Discuss](#) [Courses](#) [Practice](#) [Video](#)

## What is a destructor?

Destructor is an instance member function which is invoked automatically whenever an object is going to be destroyed. Meaning, a destructor is the last function that is going to be called before an object is destroyed.

- Destructor is also a special member function like constructor. Destructor destroys the class objects created by constructor.
- Destructor has the same name as their class name preceded by a tilde (~) symbol.
- It is not possible to define more than one destructor.
- The destructor is only one way to destroy the object created by constructor. Hence destructor can-not be overloaded.
- Destructor neither requires any argument nor returns any value.
- It is automatically called when object goes out of scope.
- Destructor releases memory space occupied by the objects created by constructor.
- In destructor, objects are destroyed in the reverse of an object creation.

The thing is to be noted here, if the object is created by using new or the constructor uses new to allocate memory which resides in the heap memory or the free store, the destructor should use delete to free the memory.

## Syntax:

```
Syntax for defining the destructor within the class  
~ <class-name>()  
{
```

```
}
```

Syntax for defining the destructor outside the class

```
<class-name> : ~ <class-name>()
```

```
{
```

```
}
```

## C++

```
// Example:
```

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

class Test
{
public:
    Test()
    {
        cout<<"\n Constructor executed";
    }

    ~Test()
    {
        cout<<"\n Destructor executed";
    }
};

main()
{
    Test t;

    return 0;
}
```

## Output

AD

```
Constructor executed
```

```
Destructor executed
```

---

## C++

```
// Example:  
  
#include<iostream>  
using namespace std;  
class Test  
{  
public:  
    Test()  
    {  
        cout<<"\n Constructor executed";  
    }  
  
    ~Test()  
    {  
        cout<<"\n Destructor executed";  
    }  
};  
  
main()  
{  
    Test t,t1,t2,t3;  
    return 0;  
}
```

## Output

```
Constructor executed  
Constructor executed  
Constructor executed  
Constructor executed  
Destructor executed  
Destructor executed  
Destructor executed  
Destructor executed
```

---

## C++

```
// Example:  
  
#include<iostream>  
using namespace std;  
int count=0;  
class Test
```

```

{
    public:
        Test()
        {
            count++;
            cout<<"\n No. of Object created:\t" << count;
        }

        ~Test()
        {
            cout<<"\n No. of Object destroyed:\t" << count;
            --count;
        }
};

main()
{
    Test t,t1,t2,t3;
    return 0;
}

```

## Output

```

No. of Object created:      1
No. of Object created:      2
No. of Object created:      3
No. of Object created:      4
No. of Object destroyed:    4
No. of Object destroyed:    3
No. of Object destroyed:    2
No. of Object destroyed:    1

```

## Properties of Destructor:

- Destructor function is automatically invoked when the objects are destroyed.
- It cannot be declared static or const.
- The destructor does not have arguments.
- It has no return type not even void.
- An object of a class with a Destructor cannot become a member of the union.
- A destructor should be declared in the public section of the class.
- The programmer cannot access the address of destructor.

## When is destructor called?

A destructor function is called automatically when the object goes out of scope:

- (1) the function ends
- (2) the program ends

(3) a block containing local variables ends

(4) a delete operator is called

**Note:** destructor can also be called explicitly for an object.

**syntax:**

`object_name.~class_name()`

### How are destructors different from a normal member function?

Destructors have same name as the class preceded by a tilde (~)

Destructors don't take any argument and don't return anything

---

## CPP

```
class String {
private:
    char* s;
    int size;

public:
    String(char*); // constructor
    ~String(); // destructor
};

String::String(char* c)
{
    size = strlen(c);
    s = new char[size + 1];
    strcpy(s, c);
}
String::~String() { delete[] s; }
```

### Can there be more than one destructor in a class?

No, there can only one destructor in a class with classname preceded by ~, no parameters and no return type.

### When do we need to write a user-defined destructor?

If we do not write our own destructor in class, compiler creates a default destructor for us. The default destructor works fine unless we have dynamically allocated memory or pointer in class. When a class contains a pointer to memory allocated in class, we should write a destructor to release memory before the class instance is destroyed. This must be done to avoid memory leak.

### Can a destructor be virtual?

Yes, In fact, it is always a good idea to make destructors virtual in base class when we have

a virtual function. See [virtual destructor](#) for more details.

You may like to take a [quiz on destructors](#).

### Related Articles :

[Constructors in C++](#)

[Virtual Destructor](#)

[Pure virtual destructor in C++](#)

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

263

## Related Articles

1. [Playing with Destructors in C++](#)
2. [C++ Interview questions based on constructors/ Destructors.](#)
3. [C++ Error – Does not name a type](#)
4. [Execution Policy of STL Algorithms in Modern C++](#)
5. [Square Meter \(Meter Squared\)](#)
6. [C++ Program To Print Pyramid Patterns](#)
7. [Jagged Arrays in C++](#)
8. [Introduction to Parallel Programming with OpenMP in C++](#)
9. [Hollow Half Pyramid Pattern Using Numbers](#)
10. [30 OOPs Interview Questions and Answers \(2023\)](#)

Previous

Next

### Article Contributed By :



GeeksforGeeks

**SALE!**  
**GeeksforGeeks Courses Upto 25% Off Enroll Now!**



Save 25% on Courses   DSA   Data Structures   Algorithms   Interview Preparation   Data Science   T

# Default Constructors in C++

Difficulty Level : Easy • Last Updated : 16 Mar, 2023

[Read](#)   [Discuss](#)   [Courses](#)   [Practice](#)   [Video](#)

A constructor without any arguments or with the default value for every argument is said to be the **Default constructor**.

A constructor that has zero parameter list or in other sense, a constructor that accept no arguments is called a zero argument constructor or default constructor.

If default constructor is not defined in the source code by the programmer, then the compiler defined the default constructor implicitly during compilation.

If the default constructor is defined explicitly in the program by the programmer, then the compiler will not defined the constructor implicitly, but it calls the constructor implicitly.

AD

## What is the significance of the default constructor?

They are used to create objects, which do not have any specific initial value.

## Is a default constructor automatically provided?

If no constructors are explicitly declared in the class, a default constructor is provided automatically by the compiler.

### Can a default constructor contain a default argument?

Yes, a constructor can contain default argument with default values for an object.

### Will there be any code inserted by the compiler to the user implemented default constructor behind the scenes?

The compiler will implicitly declare the default constructor if not provided by the programmer, will define it when in need. The compiler-defined default constructor is required to do certain initialization of class internals. It will not touch the data members or plain old data types (aggregates like an array, structures, etc...). However, the compiler generates code for the default constructor based on the situation.

Consider a class derived from another class with the default constructor, or a class containing another class object with the default constructor. The compiler needs to insert code to call the default constructors of the base class/embedded object.

---

## C++

```
// CPP program to demonstrate Default constructors
#include <iostream>
using namespace std;

class Base {
public:
    // compiler "declares" constructor
};

class A {
public:
    // User defined constructor
    A() { cout << "A Constructor" << endl; }

    // uninitialized
    int size;
};

class B : public A {
    // compiler defines default constructor of B, and
    // inserts stub to call A constructor

    // compiler won't initialize any data of A
};

class C : public A {
public:
    C()
    {

```

```

// User defined default constructor of C
// Compiler inserts stub to call A's constructor
cout << "C Constructor" << endl;

        // compiler won't initialize any data of A
    }

};

class D {
public:
    D()
    {
        // User defined default constructor of D
        // a - constructor to be called, compiler inserts
        // stub to call A constructor
        cout << "D Constructor" << endl;

        // compiler won't initialize any data of 'a'
    }

private:
    A a;
};

// Driver Code
int main()
{
    Base base;

    B b;
    C c;
    D d;

    return 0;
}

```

## Output

```

A Constructor
A Constructor
C Constructor
A Constructor
D Constructor

```

## C++

### Example:

```

#include<iostream>
using namespace std;
class student
{
    int rno;

```

```

char name[50];
double fee;
public:
student() // Explicit Default constructor
{
    cout<<"Enter the RollNo:";
    cin>>rno;
    cout<<"Enter the Name:";
    cin>>name;
    cout<<"Enter the Fee:";
    cin>>fee;
}

void display()
{
    cout<<endl<<rno<<"\t"<<name<<"\t"<<fee;
}
};

int main()
{
    student s;
    s.display();
    return 0;
}

```

There are different scenarios in which the compiler needs to insert code to ensure some necessary initialization as per language requirements. We will have them in upcoming posts. Our objective is to be aware of C++ internals, not to use them incorrectly.

---

## C++

```

// CPP code to demonstrate constructor can have default
// arguments
#include <iostream>
using namespace std;
class A {
public:
    int sum = 0;
    A(); // default constructor with no argument
    A(int a, int x = 0) // default constructor with one
                        // default argument
    {
        sum = a + x;
    }
    void print() { cout << "Sum =" << sum << endl; }
};
int main()
{
    // This construct has two arguments. Second argument is
    // initialized with a value of 0 Now we can call the
    // constructor in two possible ways.
    A obj1(10, 20);
}

```

```

A obj2(5);
obj1.print();
obj2.print();
return 0;
}

```

## Output

Sum =30  
Sum =5

**Explanation :** Here, we have a constructor with two parameter- simple parameter and one default parameter. Now, there are two ways of calling this constructor:

1. First, we can assign values to both the arguments and these values will be passed to the constructor and the default argument x with value 0 will be overridden by value passed while calling (in this case 20). Hence, code will give an output of 30 (as, sum= a+x i.e  $10+20= 30$ ).
2. Second way is to not pass any value for the default parameter. If you do so, x will take it's default value 0 as it's final value and calculate a sum of 5 (as, sum = a+x i.e  $5+0=5$ ).

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

77

## Related Articles

1. When Does Compiler Create Default and Copy Constructors in C++?

---

2. Inheritance and Constructors in Java

---

3. When are Constructors Called?

---

4. Constructors in Java

---

5. C++ | Constructors | Question 2

---

6. C++ | Constructors | Question 4

---

7. C++ | Constructors | Question 5

**SALE!**  
**GeeksforGeeks Courses Upto 25% Off Enroll Now!**



Save 25% on Courses   DSA   Data Structures   Algorithms   Interview Preparation   Data Science   T

# Private Destructor in C++

Difficulty Level : Medium • Last Updated : 17 Jan, 2023

[Read](#)   [Discuss](#)   [Courses](#)   [Practice](#)   [Video](#)

Destructors with the [access modifier](#) as private are known as Private Destructors.

Whenever we want to prevent the destruction of an object, we can make the destructor private.

## What is the use of private destructor?

Whenever we want to control the destruction of objects of a class, we make the destructor private. For dynamically created objects, it may happen that you pass a pointer to the object to a function and the function deletes the object. If the object is referred after the function call, the reference will become dangling.

## Predict the Output of the Following Programs:

AD

## CPP

```
// CPP program to illustrate  
// Private Destructor  
#include <iostream>  
using namespace std;  
  
class Test {  
private:
```

```

~Test() {}
};

int main() {}

```

The above program compiles and runs fine. Hence, we can say that: It is **not** a compiler error to create private destructors.

Now, What do you say about the below program?

---

## CPP

```

// CPP program to illustrate
// Private Destructor
#include <iostream>
using namespace std;

class Test {
private:
    ~Test() {}
};

int main() { Test t; }

```

## Output

```

prog.cpp: In function ‘int main()’:
prog.cpp:8:5: error: ‘Test::~Test()’ is private
    ~Test() {}
           ^
prog.cpp:10:19: error: within this context
    int main() { Test t; }

```

The above program fails in the compilation. The compiler notices that the local variable 't' cannot be destructed because the destructor is private.

**Now, What about the Below Program?**

---

## CPP

```

// CPP program to illustrate
// Private Destructor
#include <iostream>
using namespace std;

class Test {
private:
    ~Test() {}
};

int main() { Test* t; }

```

The above program works fine. There is no object being constructed, the program just creates a pointer of type "Test \*", so nothing is destructed.

### Next, What about the below program?

---

## CPP

```
// CPP program to illustrate
// Private Destructor
#include <iostream>
using namespace std;

class Test {
private:
    ~Test() {}
};

int main() { Test* t = new Test; }
```

The above program also works fine. When something is created using dynamic memory allocation, it is the programmer's responsibility to delete it. So compiler doesn't bother.

**In the case where the destructor is declared private, an instance of the class can also be created using the malloc() function.** The same is implemented in the below program.

---

## CPP

```
// CPP program to illustrate
// Private Destructor

#include <bits/stdc++.h>
using namespace std;

class Test {
public:
    Test() // Constructor
    {
        cout << "Constructor called\n";
    }

private:
    ~Test() // Private Destructor
    {
        cout << "Destructor called\n";
    }
};

int main()
{
    Test* t = (Test*)malloc(sizeof(Test));
    return 0;
}
```

}

The above program also works fine. However, The below program fails in the compilation. When we call delete, destructor is called.

---

## CPP

```
// CPP program to illustrate
// Private Destructor
#include <iostream>
using namespace std;

class Test {
private:
    ~Test() {}
};

// Driver Code
int main()
{
    Test* t = new Test;
    delete t;
}
```

We noticed in the above programs when a class has a private destructor, only dynamic objects of that class can be created. Following is a way to **create classes with private destructors and have a function as a friend of the class**. The function can only delete the objects.

---

## CPP

```
// CPP program to illustrate
// Private Destructor
#include <iostream>

// A class with private destructor
class Test {
private:
    ~Test() {}

public:
    friend void destructTest(Test*);
};

// Only this function can destruct objects of Test
void destructTest(Test* ptr) { delete ptr; }

int main()
{
```

```

// create an object
Test* ptr = new Test;

// destruct the object
destructTest(ptr);

return 0;
}

```

Another way to use private destructors is by using **the class instance method**.

---

## C++

```

#include <iostream>

using namespace std;

class parent {
    // private destructor
~parent() { cout << "destructor called" << endl; }

public:
    parent() { cout << "constructor called" << endl; }
    void destruct() { delete this; }
};

int main()
{
    parent* p;
    p = new parent;
    // destructor called
    p->destruct();

    return 0;
}

```

## Output

```

constructor called
destructor called

```

### Must Read: [Can a Constructor be Private in C++?](#)

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

SALE!  
GeeksforGeeks Courses Upto 25% Off Enroll Now!



Save 25% on Courses DSA Data Structures Algorithms Interview Preparation Data Science T

# Exception Handling in C++

Difficulty Level : Medium • Last Updated : 22 Jun, 2022

[Read](#) [Discuss](#) [Courses](#) [Practice](#) [Video](#)

One of the advantages of C++ over C is Exception Handling. Exceptions are runtime anomalies or abnormal conditions that a program encounters during its execution. There are two types of exceptions: a) Synchronous, b) Asynchronous (i.e., exceptions which are beyond the program's control, such as disc failure, keyboard interrupts etc.). C++ provides the following specialized keywords for this purpose:

*try*: Represents a block of code that can throw an exception.

*catch*: Represents a block of code that is executed when a particular exception is thrown.

*throw*: Used to throw an exception. Also used to list the exceptions that a function throws but doesn't handle itself.

## Why Exception Handling?

The following are the main advantages of exception handling over traditional error handling:

**1) Separation of Error Handling code from Normal Code:** In traditional error handling codes, there are always if-else conditions to handle errors. These conditions and the code to handle errors get mixed up with the normal flow. This makes the code less readable and maintainable. With try/catch blocks, the code for error handling becomes separate from the normal flow.

**2) Functions/Methods can handle only the exceptions they choose:** A function can throw many exceptions, but may choose to handle some of them. The other exceptions, which are thrown but not caught, can be handled by the caller. If the caller chooses not to catch them, then the exceptions are handled by the caller of the caller.

In C++, a function can specify the exceptions that it throws using the *throw* keyword. The caller of this function must handle the exception in some way (either by specifying it again or catching it).

AD

**3) Grouping of Error Types:** In C++, both basic types and objects can be thrown as exceptions. We can create a hierarchy of exception objects, group exceptions in namespaces or classes and categorize them according to their types.

### C++ Exceptions:

When executing C++ code, different errors can occur: coding errors made by the programmer, errors due to wrong input, or other unforeseeable things.

When an error occurs, C++ will normally stop and generate an error message. The technical term for this is: C++ will throw an exception (error).

### C++ try and catch:

Exception handling in C++ consists of three keywords: try, throw and catch:

The try statement allows you to define a block of code to be tested for errors while it is being executed.

The throw keyword throws an exception when a problem is detected, which lets us create a custom error.

The catch statement allows you to define a block of code to be executed if an error occurs in the try block.

The try and catch keywords come in pairs:

We use the try block to test some code: If the value of a variable "age" is less than 18, we will throw an exception, and handle it in our catch block.

In the catch block, we catch the error if it occurs and do something about it. The catch statement takes a single parameter. So, if the value of age is 15 and that's why we are throwing an exception of type int in the try block (age), we can pass "int myNum" as the parameter to the catch statement, where the variable "myNum" is used to output the value of age.

If no error occurs (e.g. if age is 20 instead of 15, meaning it will be greater than 18), the catch block is skipped.

## Exception Handling in C++

- 1)** The following is a simple example to show exception handling in C++. The output of the program explains the flow of execution of try/catch blocks.

### CPP

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

int main()
{
    int x = -1;

    // Some code
    cout << "Before try \n";
    try {
        cout << "Inside try \n";
        if (x < 0)
        {
            throw x;
            cout << "After throw (Never executed) \n";
        }
    }
    catch (int x ) {
        cout << "Exception Caught \n";
    }

    cout << "After catch (Will be executed) \n";
    return 0;
}
```

### Output:

```
Before try
Inside try
Exception Caught
After catch (Will be executed)
```

- 2)** There is a special catch block called the 'catch all' block, written as catch(...), that can be used to catch all types of exceptions. For example, in the following program, an int is thrown as an exception, but there is no catch block for int, so the catch(...) block will be executed.

### CPP

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

```

int main()
{
    try {
        throw 10;
    }
    catch (char *excp) {
        cout << "Caught " << excp;
    }
    catch (...) {
        cout << "Default Exception\n";
    }
    return 0;
}

```

**Output:**

Default Exception

- 3)** Implicit type conversion doesn't happen for primitive types. For example, in the following program, 'a' is not implicitly converted to int.

**CPP**

```

#include <iostream>
using namespace std;

int main()
{
    try {
        throw 'a';
    }
    catch (int x) {
        cout << "Caught " << x;
    }
    catch (...) {
        cout << "Default Exception\n";
    }
    return 0;
}

```

**Output:**

Default Exception

- 4)** If an exception is thrown and not caught anywhere, the program terminates abnormally. For example, in the following program, a char is thrown, but there is no catch block to catch the char.

## CPP

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

int main()
{
    try {
        throw 'a';
    }
    catch (int x) {
        cout << "Caught ";
    }
    return 0;
}
```

### Output:

terminate called after throwing an instance of 'char'

This application has requested the Runtime to terminate it in an unusual way. Please contact the application's support team for more information.

We can change this abnormal termination behavior by [writing our own unexpected function](#).

- 5) A derived class exception should be caught before a base class exception. See [this](#) for more details.
- 6) Like Java, the C++ library has a [standard exception class](#) which is the base class for all standard exceptions. All objects thrown by the components of the standard library are derived from this class. Therefore, all standard exceptions can be caught by catching this type
- 7) Unlike Java, in C++, all exceptions are unchecked, i.e., the compiler doesn't check whether an exception is caught or not (See [this](#) for details). So, it is not necessary to specify all uncaught exceptions in a function declaration. Although it's a recommended practice to do so. For example, the following program compiles fine, but ideally the signature of fun() should list the unchecked exceptions.

## CPP

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

// This function signature is fine by the compiler, but not recommended.
```

```
// Ideally, the function should specify all uncaught exceptions and function
// signature should be "void fun(int *ptr, int x) throw (int *, int)"
void fun(int *ptr, int x)
{
    if (ptr == NULL)
        throw ptr;
    if (x == 0)
        throw x;
    /* Some functionality */
}

int main()
{
    try {
        fun(NULL, 0);
    }
    catch(...) {
        cout << "Caught exception from fun()";
    }
    return 0;
}
```

**Output:**

Caught exception from fun()

A better way to write the above code:

---

**CPP**

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

// Here we specify the exceptions that this function
// throws.
void fun(int *ptr, int x) throw (int *, int) // Dynamic Exception specification
{
    if (ptr == NULL)
        throw ptr;
    if (x == 0)
        throw x;
    /* Some functionality */
}

int main()
{
    try {
        fun(NULL, 0);
    }
    catch(...) {
        cout << "Caught exception from fun()";
    }
    return 0;
}
```

```

}
```

**Note :** The use of Dynamic Exception Specification has been deprecated since C++11. One of the reasons for it may be that it can randomly abort your program. This can happen when you throw an exception of another type which is not mentioned in the dynamic exception specification. Your program will abort itself because in that scenario, it calls (indirectly) `terminate()`, which by default calls `abort()`.

### Output:

```
Caught exception from fun()
```

- 8)** In C++, try/catch blocks can be nested. Also, an exception can be re-thrown using "throw;".
- 

## CPP

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

int main()
{
    try {
        try {
            throw 20;
        }
        catch (int n) {
            cout << "Handle Partially ";
            throw; // Re-throwing an exception
        }
    }
    catch (int n) {
        cout << "Handle remaining ";
    }
    return 0;
}
```

### Output:

```
Handle Partially Handle remaining
```

A function can also re-throw a function using the same "throw;" syntax. A function can handle a part and ask the caller to handle the remaining.

- 9)** When an exception is thrown, all objects created inside the enclosing try block are destroyed before the control is transferred to the catch block.
-

## CPP

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

class Test {
public:
    Test() { cout << "Constructor of Test " << endl; }
    ~Test() { cout << "Destructor of Test " << endl; }
};

int main()
{
    try {
        Test t1;
        throw 10;
    }
    catch (int i) {
        cout << "Caught " << i << endl;
    }
}
```

### Output:

```
Constructor of Test
Destructor of Test
Caught 10
```

**10)** You may like to try [Quiz on Exception Handling in C++](#).

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

263

### Related Articles

1. [Comparison of Exception Handling in C++ and Java](#)

---
2. [C++ | Exception Handling | Question 1](#)

---
3. [C++ | Exception Handling | Question 3](#)

---
4. [C++ | Exception Handling | Question 4](#)

---

**SALE!**  
**GeeksforGeeks Courses Upto 25% Off Enroll Now!**



Save 25% on Courses DSA Data Structures Algorithms Interview Preparation Data Science T

# Stack Unwinding in C++

Difficulty Level : Medium • Last Updated : 25 Nov, 2021

[Read](#) [Discuss](#) [Courses](#) [Practice](#) [Video](#)

**Stack Unwinding** is the process of removing function entries from function call stack at run time. The local objects are destroyed in reverse order in which they were constructed.

Stack Unwinding is generally related to [Exception Handling](#). In C++, when an exception occurs, the function call stack is linearly searched for the exception handler, and all the entries before the function with exception handler are removed from the function call stack. So, exception handling involves Stack Unwinding if an exception is not handled in the same function (where it is thrown). Basically, Stack unwinding is a process of calling the destructors (whenever an exception is thrown) for all the automatic objects constructed at run time.

**For example, the output of the following program is:**

## CPP

```
// CPP Program to demonstrate Stack Unwinding
#include <iostream>
using namespace std;

// A sample function f1() that throws an int exception
void f1() throw(int)
{
    cout << "\n f1() Start ";
    throw 100;
    cout << "\n f1() End ";
}

// Another sample function f2() that calls f1()
void f2() throw(int)
{
    cout << "\n f2() Start ";
    f1();
```

```

        cout << "\n f2() End ";
    }

// Another sample function f3() that calls f2() and handles
// exception thrown by f1()
void f3()
{
    cout << "\n f3() Start ";
    try {
        f2();
    }
    catch (int i) {
        cout << "\n Caught Exception: " << i;
    }
    cout << "\n f3() End";
}

// Driver Code
int main()
{
    f3();

    getchar();
    return 0;
}

```

## Output

```

f3() Start
f2() Start
f1() Start
Caught Exception: 100
f3() End

```

## Explanation:

AD

- When f1() throws exception, its entry is removed from the function call stack, because f1() doesn't contain exception handler for the thrown exception, then next entry in call stack is looked for exception handler.
- The next entry is f2(). Since f2() also doesn't have a handler, its entry is also removed from the function call stack.

- The next entry in the function call stack is f3(). Since f3() contains an exception handler, the catch block inside f3() is executed, and finally, the code after the catch block is executed.

Note that the following lines inside f1() and f2() are not executed at all.

```
cout<<"\n f1() End "; // inside f1()
```

```
cout<<"\n f2() End "; // inside f2()
```

If there were some local class objects inside f1() and f2(), destructors for those local objects would have been called in the Stack Unwinding process.

**Note:** Stack Unwinding also happens in Java when exception is not handled in same function.

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

61

## Related Articles

1. How to detect Stack Unwinding in a Destructor in C++?
2. stack empty() and stack size() in C++ STL
3. stack swap() in C++ STL
4. stack top() in C++ STL
5. Stack push() and pop() in C++ STL
6. stack emplace() in C++ STL
7. Implementing Stack Using Class Templates in C++
8. How to implement a Stack using list in C++ STL
9. Stack-buffer based STL allocator

SALE!

GeeksforGeeks Courses Upto 25% Off Enroll Now!



Save 25% on Courses

DSA

Data Structures

Algorithms

Interview Preparation

Data Science

T

# Catching Base and Derived Classes as Exceptions in C++ and Java

Difficulty Level : Easy • Last Updated : 02 Mar, 2023

[Read](#)[Discuss](#)[Courses](#)[Practice](#)[Video](#)

An Exception is an unwanted error or hurdle that a program throws while compiling. There are various methods to handle an exception which is termed exceptional handling.

Let's discuss what is Exception Handling and how we catch base and derived classes as an exception in C++:

- If both base and derived classes are caught as exceptions, then the catch block of the derived class must appear before the base class.
- If we put the base class first then the derived class catch block will never be reached.  
For example, the following **C++** code prints "**Caught Base Exception**".

## C++

```
// C++ Program to demonstrate a
// Catching Base Exception
#include <iostream>
using namespace std;

class Base {
};
class Derived : public Base {
};
int main()
{
    Derived d;
    // Some other functionalities
    try {
        // Monitored code
        throw d;
    }
```

```

catch (Base b) {
    cout << "Caught Base Exception";
}
catch (Derived d) {
    // This 'catch' block is NEVER executed
    cout << "Caught Derived Exception";
}
getchar();
return 0;
}

```

## Output

Caught Base Exception

*The **output** of the above **C++** code:*

```

prog.cpp: In function ‘int main()’:
prog.cpp:20:5: warning: exception of type ‘Derived’ will be caught
    catch (Derived d) {
        ^
prog.cpp:17:5: warning:     by earlier handler for ‘Base’
    catch (Base b) {

```

In the above C++ code, if we change the order of catch statements then both catch statements become reachable.

AD

*Following is the modified program and it prints “Caught Derived Exception”*

---

## C++

```

// C++ Program to demonstrate a catching of
// Derived exception and printing it successfully
#include <iostream>
using namespace std;

class Base {};
class Derived : public Base {};
int main()

```

```

{
    Derived d;
    // Some other functionalities
    try {
        // Monitored code
        throw d;
    }
    catch (Derived d) {
        cout << "Caught Derived Exception";
    }
    catch (Base b) {
        cout << "Caught Base Exception";
    }
    getchar(); // To read the next character
    return 0;
}

```

## Output

Caught Derived Exception

## Output:

Caught Derived Exception

In java, catching a base class exception before derived is not allowed by the compiler itself.  
In C++, the compiler might give a warning about it but compiles the code.

*For example, the following Java code fails in compilation with the error message  
"exception Derived has already been caught"*

## Java

```

// Java Program to demonstrate
// the error filename Main.java
class Base extends Exception {
}
class Derived extends Base {
}
public class Main {
    public static void main(String args[])
    {
        try {
            throw new Derived();
        }
        catch (Base b) {
        }
        catch (Derived d) {
        }
    }
}

```

}

**Error:**

```
prog.java:11: error: exception Derived has already been caught
    catch(Derived d) {}
```

In both C++ and Java, you can catch both base and derived classes as exceptions. This is useful when you want to catch multiple exceptions that may have a common base class.

In C++, you can catch base and derived classes as exceptions using the catch block. When you catch a base class, it will also catch any derived classes of that base class. Here's an example:

**C++**

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

class BaseException : public exception {
public:
    virtual const char* what() const throw() {
        return "Base exception";
    }
};

class DerivedException : public BaseException {
public:
    virtual const char* what() const throw() {
        return "Derived exception";
    }
};

int main() {
    try {
        // code that might throw exceptions
        throw DerivedException();
    } catch (BaseException& e) {
        cout << "Caught exception: " << e.what() << endl;
    }
    return 0;
}
```

**Output**

Caught exception: Derived exception

In this example, a `BaseException` class is defined and a `DerivedException` class is derived from it. In the `main()` function, a `DerivedException` object is thrown. The catch block catches any `BaseException` object or derived object, and prints a message to the console indicating which exception was caught.

In Java, you can catch base and derived classes as exceptions using the catch block with multiple catch clauses. When you catch a base class, it will also catch any derived classes of that base class.

Here's an example:

## Java

```
class BaseException extends Exception {
    public BaseException() {
        super("Base exception");
    }
}

class DerivedException extends BaseException {
    public DerivedException() {
        super("Derived exception");
    }
}

public class ExceptionExample {
    public static void main(String[] args) {
        try {
            // code that might throw exceptions
            throw new DerivedException();
        } catch (DerivedException e) {
            System.out.println("Caught derived exception: " + e.getMessage());
        } catch (BaseException e) {
            System.out.println("Caught base exception: " + e.getMessage());
        }
    }
}
```

OUTPUT:

Caught derived exception: Derived exception

**SALE!**  
**GeeksforGeeks Courses Upto 25% Off Enroll Now!**



Save 25% on Courses DSA Data Structures Algorithms Interview Preparation Data Science T

# Catch block and type conversion in C++

Difficulty Level : Medium • Last Updated : 12 Nov, 2021

[Read](#) [Discuss](#) [Courses](#) [Practice](#) [Video](#)

Predict the output of following C++ program.

## C++

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

int main()
{
    try
    {
        throw 'x';
    }
    catch(int x)
    {
        cout << " Caught int " << x;
    }
    catch(...)
    {
        cout << "Default catch block";
    }
}
```

## Output:

Default catch block

In the above program, a character 'x' is thrown and there is a catch block to catch an int. One might think that the int catch block could be matched by considering ASCII value of 'x'. But

such conversions are not performed for catch blocks. Consider the following program as another example where conversion constructor is not called for thrown object.

---

## C++

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

class MyExcept1 {};

class MyExcept2
{
public:

    // Conversion constructor
    MyExcept2 (const MyExcept1 &e )
    {
        cout << "Conversion constructor called";
    }
};

int main()
{
    try
    {
        MyExcept1 myexp1;
        throw myexp1;
    }
    catch(MyExcept2 e2)
    {
        cout << "Caught MyExcept2 " << endl;
    }
    catch(...)
    {
        cout << " Default catch block " << endl;
    }
    return 0;
}
```

## Output:

AD

## Default catch block

As a side note, the derived type objects are converted to base type when a derived object is thrown and there is a catch block to catch base type. See [this GFact](#) for more details.

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

23

## Related Articles

1. [Java Multiple Catch Block](#)
2. [Difference between Type Casting and Type Conversion](#)
3. [Conversion of Struct data type to Hex String and vice versa](#)
4. [Type Conversion in C++](#)
5. [Implicit Type Conversion in C with Examples](#)
6. [Type Conversion in C](#)
7. [Flow control in try catch finally in Java](#)
8. [Ask\(\), Wait\(\) and Answer\(\) Block in Scratch Programming](#)
9. [Where is an object stored if it is created inside a block in C++?](#)
10. [C++ Program for Block swap algorithm for array rotation](#)

Previous

Next

## Article Contributed By :



GeeksforGeeks

**SALE!**  
**GeeksforGeeks Courses Upto 25% Off Enroll Now!**



Save 25% on Courses DSA Data Structures Algorithms Interview Preparation Data Science T

# Exception Handling and Object Destruction in C++

Difficulty Level : Easy • Last Updated : 20 Jan, 2023

Read Discuss Courses Practice Video

An exception is termed as an unwanted error that arises during the runtime of the program. The practice of separating the anomaly-causing program/code from the rest of the program/code is known as [Exception Handling](#).

An object is termed as an instance of the class which has the same name as that of the class. A [destructor](#) is a member function of a class that has the same name as that of the class but is preceded by a ‘~’ (tilde) sign, also it is automatically called after the scope of the code runs out. The practice of pulverizing or demolition of the existing object memory is termed *object destruction*.

In other words, the class of the program never holds any kind of memory or storage, it is the object which holds the memory or storage and to deallocate/destroy the memory of created object we use destructors.

## For Example:

### CPP

```
// C++ Program to show the sequence of calling  
// Constructors and destructors  
#include <iostream>  
using namespace std;  
  
// Initialization of class  
class Test {  
public:  
    // Constructor of class  
    Test()  
    {
```

```
cout << "Constructing an object of class Test "
     << endl;
}

// Destructor of class
~Test()
{
    cout << "Destructing the object of class Test "
         << endl;
}
};

int main()
{
    try {
        // Calling the constructor
        Test t1;
        throw 10;

    } // Destructor is being called here
      // Before the 'catch' statement
    catch (int i) {
        cout << "Caught " << i << endl;
    }
}
```

## Output:

AD

```
Constructing an object of class Test
Destructing the object of class Test
Caught 10
```

When an exception is thrown, destructors of the objects (whose scope ends with the try block) are automatically called before the catch block gets executed. That is why the above program prints "**Destructing an object of Test**" before "**Caught 10**".

## What Happens When an Exception is Thrown From a Constructor?

### Example:

## CPP

```
// C++ Program to show what really happens
// when an exception is thrown from
// a constructor
#include <iostream>
using namespace std;

class Test1 {
public:
    // Constructor of the class
    Test1()
    {
        cout << "Constructing an Object of class Test1"
            << endl;
    }
    // Destructor of the class
    ~Test1()
    {
        cout << "Destructuring an Object the class Test1"
            << endl;
    }
};

class Test2 {
public:
    // Following constructor throws
    // an integer exception
    Test2() // Constructor of the class
    {
        cout << "Constructing an Object of class Test2"
            << endl;
        throw 20;
    }
    // Destructor of the class
    ~Test2()
    {
        cout << "Destructuring the Object of class Test2"
            << endl;
    }
};

int main()
{
    try {
        // Constructed and destructured
        Test1 t1;

        // Partially constructed
        Test2 t2;

        // t3 is not constructed as
        // this statement never gets executed
        Test1 t3; // t3 is not called as t2 is
                  // throwing/returning 'int' argument which
    }
}
```

```

        // is not accepted
        // is the class test1'
    }
catch (int i) {
    cout << "Caught " << i << endl;
}
}

```

**Output:**

Constructing an Object of class Test1  
 Constructing an Object of class Test2  
 Destructuring an Object the class Test1  
 Caught 20

Destructors are only called for the completely constructed objects. When the constructor of an object throws an exception, the destructor for that object is not called.

***Predict the output of the following program:***

---

**CPP**

```

// C++ program to show how many times
// Constructors and destructors are called
#include <iostream>
using namespace std;

class Test {
    static int count; // Used static to initialise the scope
                      // Of 'count' till lifetime
    int id;

public:
    // Constructor
    Test()
    {
        count++;
        id = count;
        cout << "Constructing object number " << id << endl;
        if (id == 4)
            throw 4;
    }
    // Destructor
    ~Test()
    {
        cout << "Destructuring object number " << id << endl;
    }
};

int Test::count = 0;

```

```
// Source code
int main()
{
    try {
        Test array[5];
    }
    catch (int i) {
        cout << "Caught " << i << endl;
    }
}
```

## Output:

```
Constructing object number 1
Constructing object number 2
Constructing object number 3
Constructing object number 4
Destructing object number 3
Destructing object number 2
Destructing object number 1
Caught 4
```

40

## Related Articles

1. Virtual destruction using shared\_ptr in C++

---
2. Comparison of Exception Handling in C++ and Java

---
3. Top C++ Exception Handling Interview Questions and Answers

---
4. Exception Handling in C++

---
5. C++ | Exception Handling | Question 1

---
6. C++ | Exception Handling | Question 3

---
7. C++ | Exception Handling | Question 4

---
8. C++ | Exception Handling | Question 5

---
9. C++ | Exception Handling | Question 6

---

**SALE!**  
**GeeksforGeeks Courses Upto 25% Off Enroll Now!**



Save 25% on Courses DSA Data Structures Algorithms Interview Preparation Data Science T

# File Handling through C++ Classes

Difficulty Level : Medium • Last Updated : 02 Nov, 2022

[Read](#) [Discuss](#) [Courses](#) [Practice](#) [Video](#)

File handling is used to store data permanently in a computer. Using file handling we can store our data in secondary memory (Hard disk).

How to achieve the File Handling

For achieving file handling we need to follow the following steps:-

STEP 1-Naming a file

STEP 2-Opening a file

STEP 3-Writing data into the file

STEP 4-Reading data from the file

STEP 5-Closing a file.

## Streams in C++ :-

We give input to the executing program and the execution program gives back the output. The sequence of bytes given as input to the executing program and the sequence of bytes that comes as output from the executing program are called stream. In other words, streams are nothing but the flow of data in a sequence.

The input and output operation between the executing program and the devices like keyboard and monitor are known as "console I/O operation". The input and output operation between the executing program and files are known as "disk I/O operation".

## Classes for File stream operations :-

The I/O system of C++ contains a set of classes which define the file handling methods. These include ifstream, ofstream and fstream classes. These classes are derived from fstream and from the corresponding iostream class. These classes, designed to manage

the disk files, are declared in `fstream` and therefore we must include this file in any program that uses files.

#### 1. ios:-

AD

- `ios` stands for input output stream.
- This class is the base class for other classes in this class hierarchy.
- This class contains the necessary facilities that are used by all the other derived classes for input and output operations.

#### 2. istream:-

- `istream` stands for input stream.
- This class is derived from the class '`ios`'.
- This class handle input stream.
- The extraction operator(`>>`) is overloaded in this class to handle input streams from files to the program execution.
- This class declares input functions such as `get()`, `getline()` and `read()`.

#### 3. ostream:-

- `ostream` stands for output stream.
- This class is derived from the class '`ios`'.
- This class handle output stream.
- The insertion operator(`<<`) is overloaded in this class to handle output streams to files from the program execution.
- This class declares output functions such as `put()` and `write()`.

#### 4. streambuf:-

- This class contains a pointer which points to the buffer which is used to manage the input and output streams.

#### 5. fstreambase:-

- This class provides operations common to the file streams. Serves as a base for `fstream`, `ifstream` and `ofstream` class.
- This class contains `open()` and `close()` function.

#### 6. `ifstream`:-

- This class provides input operations.
- It contains `open()` function with default input mode.
- Inherits the functions `get()`, `getline()`, `read()`, `seekg()` and `tellg()` functions from the `istream`.

#### 7. `ofstream`:-

- This class provides output operations.
- It contains `open()` function with default output mode.
- Inherits the functions `put()`, `write()`, `seekp()` and `tellp()` functions from the `ostream`.

#### 8. `fstream`:-

- This class provides support for simultaneous input and output operations.
- Inherits all the functions from `istream` and `ostream` classes through `iostream`.

#### 9. `filebuf`:-

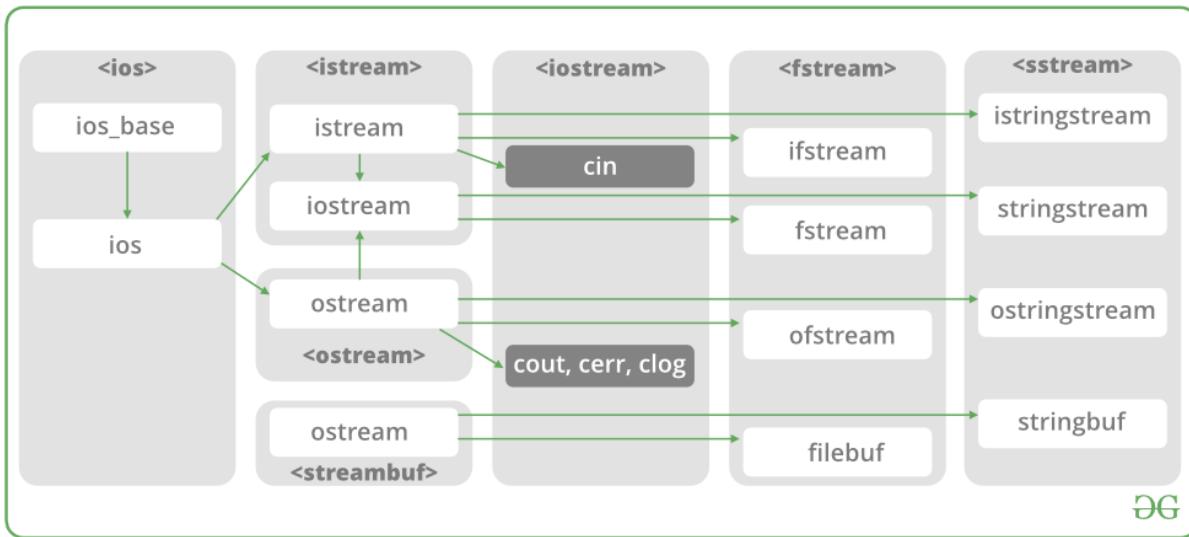
- Its purpose is to set the file buffers to read and write.
- We can also use file buffer member function to determine the length of the file.

In C++, files are mainly dealt by using three classes `fstream`, `ifstream`, `ofstream` available in `fstream` headerfile.

**ofstream:** Stream class to write on files

**ifstream:** Stream class to read from files

**fstream:** Stream class to both read and write from/to files.



Now the first step to open the particular file for read or write operation. We can open file by

1. passing file name in constructor at the time of object creation
2. using the open method

**For e.g.**

### ***Open File by using constructor***

```
ifstream (const char* filename, ios_base::openmode mode = ios_base::in);
ifstream fin(filename, openmode) by default openmode = ios::in
ifstream fin("filename");
```

### ***Open File by using open method***

*Calling of default constructor*

```
ifstream fin;
fin.open(filename, openmode)
fin.open("filename");
```

### **Modes :**

<b>Member Constant</b>	<b>Stands For</b>	<b>Access</b>
in *	input	File open for reading: the internal stream buffer supports input operations.

<b>Member Constant</b>	<b>Stands For</b>	<b>Access</b>
out	output	File open for writing: the internal stream buffer supports output operations.
binary	binary	Operations are performed in binary mode rather than text.
ate	at end	The output position starts at the end of the file.
app	append	All output operations happen at the end of the file, appending to its existing contents.
trunc	truncate	Any contents that existed in the file before it is open are discarded.

### Default Open Modes :

ifstream	ios::in
ofstream	ios::out
fstream	ios::in   ios::out

**Problem Statement :** To read and write a File in C++.

### Examples:

**Input :**

```
Welcome in GeeksforGeeks. Best way to learn things.
```

-1

**Output :**

```
Welcome in GeeksforGeeks. Best way to learn things.
```

Recommended: Please try your approach on [{IDE}](#) first, before moving on to the solution.

Below is the implementation by using **ifstream & ofstream classes**.

## C++

```
/* File Handling with C++ using ifstream & ofstream class object*/
/* To write the Content in File*/
/* Then to read the content of file*/
#include <iostream>

/* fstream header file for ifstream, ofstream,
   fstream classes */
#include <fstream>

using namespace std;

// Driver Code
int main()
{
    // Creation of ofstream class object
    ofstream fout;

    string line;

    // by default ios::out mode, automatically deletes
    // the content of file. To append the content, open in ios::app
    // fout.open("sample.txt", ios::app)
    fout.open("sample.txt");

    // Execute a loop If file successfully opened
    while (fout) {

        // Read a Line from standard input
        getline(cin, line);

        // Press -1 to exit
        if (line == "-1")
            break;

        // Write line in file
        fout << line << endl;
    }

    // Close the File
    fout.close();

    // Creation of ifstream class object to read the file
    ifstream fin;

    // by default open mode = ios::in mode
    fin.open("sample.txt");

    // Execute a loop until EOF (End of File)
    while (getline(fin, line)) {

        // Print line (read from file) in Console
        cout << line << endl;
    }
}
```

```

    // Close the file
    fin.close();

    return 0;
}

```

Below is the implementation by using **fstream class**.

---

## C++

```

/* File Handling with C++ using fstream class object */
/* To write the Content in File */
/* Then to read the content of file*/
#include <iostream>

/* fstream header file for ifstream, ofstream,
   fstream classes */
#include <fstream>

using namespace std;

// Driver Code
int main()
{
    // Creation of fstream class object
    fstream fio;

    string line;

    // by default openmode = ios::in|ios::out mode
    // Automatically overwrites the content of file, To append
    // the content, open in ios::app
    // fio.open("sample.txt", ios::in|ios::out|ios::app)
    // ios::trunc mode delete all content before open
    fio.open("sample.txt", ios::trunc | ios::out | ios::in);

    // Execute a loop If file successfully Opened
    while (fio) {

        // Read a Line from standard input
        getline(cin, line);

        // Press -1 to exit
        if (line == "-1")
            break;

        // Write line in file
        fio << line << endl;
    }

    // Execute a loop until EOF (End of File)
    // point read pointer at beginning of file
    fio.seekg(0, ios::beg);
}

```

```

while (fio) {

    // Read a Line from File
    getline(fio, line);

    // Print line in Console
    cout << line << endl;
}

// Close the file
fio.close();

return 0;
}

```

---

## C++

Q: write a single file handling program in c++ to reading and writing data on a file.

```

#include<iostream>
#include<fstream>

using namespace std;
main()
{
    int rno,fee;
    char name[50];

    cout<<"Enter the Roll Number:";
    cin>>rno;

    cout<<"\nEnter the Name:";
    cin>>name;

    cout<<"\nEnter the Fee:";
    cin>>fee;

    ofstream fout("d:/student.doc");

    fout<<rno<<"\t"<<name<<"\t"<<fee;    //write data to the file student

    fout.close();

    ifstream fin("d:/student.doc");

    fin>>rno>>name>>fee;    //read data from the file student

    fin.close();

    cout<<endl<<rno<<"\t"<<name<<"\t"<<fee;

    return 0;
}

```

## C++

```
// Q: WA C++ file handling program to read data from the file called student.doc

#include<iostream>
#include<fstream>

using namespace std;

main()
{
    int rno,fee;
    char name[50];

    ifstream fin("d:/student.doc");

    fin>>rno>>name>>fee;    //read data from the file student

    fin.close();

    cout<<endl<<rno<<"\t"<<name<<"\t"<<fee;

    return 0;
}
```

193

## Related Articles

1. Exception Handling using classes in C++
2. Set Position with seekg() in C++ File Handling
3. tellp() in file handling with c++ with example
4. How to work with file handling in C++
5. Error handling during file operations in C/C++
6. Contact Book in C++ using File Handling
7. ATM using file handling in C++



SALE!

GeeksforGeeks Courses Upto 25% Off Enroll Now!

[Save 25% on Courses](#) DSA Data Structures Algorithms Interview Preparation Data Science T

# Read/Write Class Objects from/to File in C++

Difficulty Level : Medium • Last Updated : 24 Jan, 2023

[Read](#) [Discuss](#) [Courses](#) [Practice](#) [Video](#)

Given a file "Input.txt" in which every line has values same as instance variables of a class.

Read the values into the class's object and do necessary operations.

## Theory :

The data transfer is usually done using '>>' and '<<' operators. But if you have a class with 4 data members and want to write all 4 data members from its object directly to a file or vice-versa, we can do that using following syntax :

### To write object's data members in a file :

```
// Here file_obj is an object of ofstream  
file_obj.write((char *) & class_obj, sizeof(class_obj));
```

### To read file's data members into an object :

```
// Here file_obj is an object of ifstream  
file_obj.read((char *) & class_obj, sizeof(class_obj));
```

## Examples:

### Input :

Input.txt :

Michael 19 1806

Kemp 24 2114

Terry 21 2400

Operation : Print the name of the highest

rated programmer.

#### Output :

Terry

---

## C++

```
// C++ program to demonstrate read/write of class
// objects in C++.
#include <iostream>
#include <fstream>
using namespace std;

// Class to define the properties
class Contestant {
public:
    // Instance variables
    string Name;
    int Age, Ratings;

    // Function declaration of input() to input info
    int input();

    // Function declaration of output_highest_rated() to
    // extract info from file Data Base
    int output_highest_rated();
};

// Function definition of input() to input info
int Contestant::input()
{
    // Object to write in file
    ofstream file_obj;

    // Opening file in append mode
    file_obj.open("Input.txt", ios::app);

    // Object of class contestant to input data in file
    Contestant obj;

    // Feeding appropriate data in variables
    string str = "Michael";
    int age = 18, ratings = 2500;

    // Assigning data into object
    obj.Name = str;
    obj.Age = age;
    obj.Ratings = ratings;

    // Writing the object's data in file
    file_obj.write((char*)&obj, sizeof(obj));

    // Feeding appropriate data in variables
```

```
str = "Terry";
age = 21;
ratings = 3200;

// Assigning data into object
obj.Name = str;
obj.Age = age;
obj.Ratings = ratings;

// Writing the object's data in file
file_obj.write((char*)&obj, sizeof(obj));

//close the file
//It's always a good practice to close the file after opening them
file_obj.close();

return 0;
}

// Function definition of output_highest_rated() to
// extract info from file Data Base
int Contestant::output_highest_rated()
{
    // Object to read from file
    ifstream file_obj;

    // Opening file in input mode
    file_obj.open("Input.txt", ios::in);

    // Object of class contestant to input data in file
    Contestant obj;

    // Reading from file into object "obj"
    file_obj.read((char*)&obj, sizeof(obj));

    // max to store maximum ratings
    int max = 0;

    // Highest_rated stores the name of highest rated contestant
    string Highest_rated;

    // Checking till we have the feed
    while (!file_obj.eof()) {
        // Assigning max ratings
        if (obj.Ratings > max) {
            max = obj.Ratings;
            Highest_rated = obj.Name;
        }
    }

    // Checking further
    file_obj.read((char*)&obj, sizeof(obj));
}

// close the file.
//It's always a good practice to close the file after opening them
file_obj.close();
```

```
// Output is the highest rated contestant
cout << Highest_rated;
return 0;
}

// Driver code
int main()
{
    // Creating object of the class
Contestant object;

    // Inputting the data
object.input();

    // Extracting the max rated contestant
object.output_highest_rated();

    return 0;
}
```

## Output:

Terry

This article is contributed by [Rohit Thapliyal](#). If you like GeeksforGeeks and would like to contribute, you can also write an article using [write.geeksforgeeks.org](https://write.geeksforgeeks.org) or mail your article to review-team@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

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

AD

31

## Related Articles

1. How to make a C++ class whose objects can only be dynamically allocated?

**SALE!**  
**GeeksforGeeks Courses Upto 25% Off Enroll Now!**



Save 25% on Courses DSA Data Structures Algorithms Interview Preparation Data Science T

# C++ program to create a file

Difficulty Level : Medium • Last Updated : 28 Sep, 2018

[Read](#) [Discuss](#) [Courses](#) [Practice](#) [Video](#)

**Problem Statement:** Write a C++ program to create a file using file handling and check whether the file is created successfully or not. If a file is created successfully then it should print "File Created Successfully" otherwise should print some error message.

**Approach:** Declare a stream class file and open that text file in writing mode. If the file is not present then it creates a new text file. Now check if the file does not exist or not created then return false otherwise return true.

**Below is the program to create a file:**

```
// C++ implementation to create a file
#include <bits/stdc++.h>
using namespace std;

// Driver code
int main()
{
    // fstream is Stream class to both
    // read and write from/to files.
    // file is object of fstream class
    fstream file;

    // opening file "Gfg.txt"
    // in out(write) mode
    // ios::out Open for output operations.
    file.open("Gfg.txt",ios::out);

    // If no file is created, then
    // show the error message.
    if(!file)
    {
        cout<<"Error in creating file!!!";
        return 0;
    }
}
```

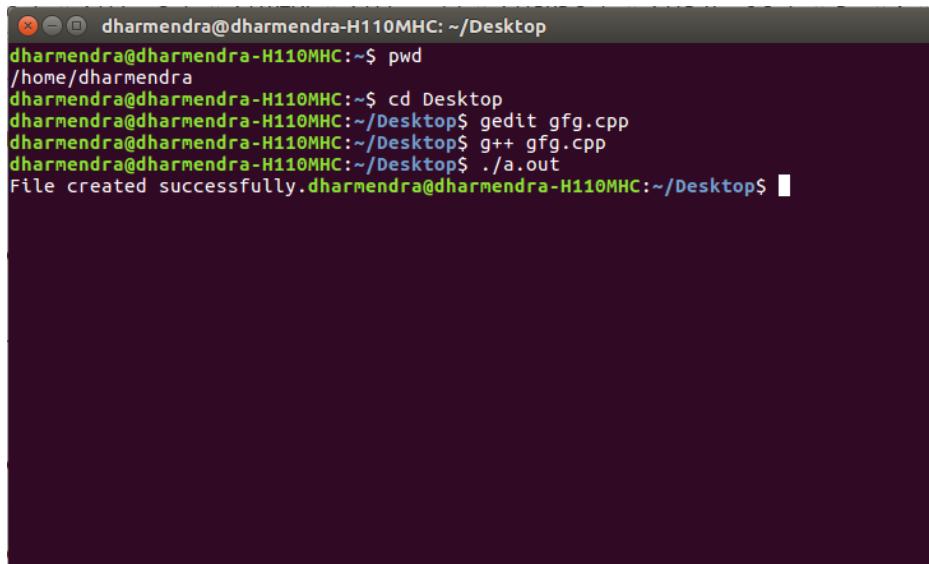
```
}

cout<<"File created successfully.";

// closing the file.
// The reason you need to call close()
// at the end of the loop is that trying
// to open a new file without closing the
// first file will fail.
file.close();

return 0;
}
```

## Output:

A screenshot of a terminal window titled "darmendra@darmendra-H110MHC: ~/Desktop". The window shows the following command-line session:

```
darmendra@darmendra-H110MHC:~/Desktop
darmendra@darmendra-H110MHC:~$ pwd
/home/darmendra
darmendra@darmendra-H110MHC:~$ cd Desktop
darmendra@darmendra-H110MHC:~/Desktop$ gedit ofg.cpp
darmendra@darmendra-H110MHC:~/Desktop$ g++ ofg.cpp
darmendra@darmendra-H110MHC:~/Desktop$ ./a.out
File created successfully.darmendra@darmendra-H110MHC:~/Desktop$
```

The terminal has a dark background with light-colored text. The cursor is visible at the end of the last command.

AD

24

## Related Articles

1. [How to create Binary File from the existing Text File?](#)
2. [C program to copy contents of one file to another file](#)
3. [C++ Program to Copy One File into Another File](#)

**SALE!**  
**GeeksforGeeks Courses Upto 25% Off Enroll Now!**



Save 25% on Courses DSA Data Structures Algorithms Interview Preparation Data Science T

# CSV file management using C++

Difficulty Level : Medium • Last Updated : 25 Jun, 2020

[Read](#) [Discuss](#) [Courses](#) [Practice](#) [Video](#)

**CSV** is a simple file format used to store tabular data such as a spreadsheet or a database. CSV stands for **Comma Separated Values**. The data fields in a CSV file are separated/delimited by a comma (',') and the individual rows are separated by a newline ('\n'). CSV File management in C++ is similar to text-type file management, except for a few modifications.

This article discusses about how to **create, update and delete records** in a CSV file:

**Note:** Here, a reportcard.csv file has been created to store the student's roll number, name and marks in math, physics, chemistry and biology.

## 1. Create operation:

The create operation is similar to creating a text file, i.e. input data from the user and write it to the csv file using the file pointer and appropriate delimiters(',') between different columns and '\n' after the end of each row.

AD

## CREATE

```
void create()
{
    // file pointer
```

```

fstream fout;

// opens an existing csv file or creates a new file.
fout.open("reportcard.csv", ios::out | ios::app);

cout << "Enter the details of 5 students:"
    << " roll name maths phy chem bio";
<< endl;

int i, roll, phy, chem, math, bio;
string name;

// Read the input
for (i = 0; i < 5; i++) {

    cin >> roll
        >> name
        >> math
        >> phy
        >> chem
        >> bio;

    // Insert the data to file
    fout << roll << ", "
        << name << ", "
        << math << ", "
        << phy << ", "
        << chem << ", "
        << bio
        << "\n";
}
}

```

## Output:

	roll	name	maths	phy	chem	bio
1	1	rahul	100	90	75	100
2	2	kaushik	90	80	50	90
3	3	nayonika	90	70	95	86
4	4	simran	55	85	70	70
5	5	sumana	65	90	95	100
6						

## 2. Read a particular record:

In reading a CSV file, the following approach is implemented:-

1. Using `getline()`, file pointer and '\n' as the delimiter, read an entire row and store it in a string variable.

2. Using stringstream, separate the row into words.
3. Now using getline(), the stringstream pointer and ',' as the delimiter, read every word in the row, store it in a string variable and push that variable to a string vector.
4. Retrieve a required column data through row[index]. Here, row[0] always stores the roll number of a student, so compare row[0] with the roll number input by the user, and if it matches, display the details of the student and break from the loop.

**Note:** Here, since whatever data reading from the file, is stored in string format, so always convert string to the required datatype before comparing or calculating, etc.

---

## READ

```
void read_record()
{
    // File pointer
    fstream fin;

    // Open an existing file
    fin.open("reportcard.csv", ios::in);

    // Get the roll number
    // of which the data is required
    int rollnum, roll2, count = 0;
    cout << "Enter the roll number "
        << "of the student to display details: ";
    cin >> rollnum;

    // Read the Data from the file
    // as String Vector
    vector<string> row;
    string line, word, temp;

    while (fin >> temp) {

        row.clear();

        // read an entire row and
        // store it in a string variable 'line'
        getline(fin, line);

        // used for breaking words
        stringstream s(line);

        // read every column data of a row and
        // store it in a string variable, 'word'
        while (getline(s, word, ',')) {

            // add all the column data
            // of a row to a vector
            row.push_back(word);
        }
    }
}
```

```

// convert string to integer for comparision
roll2 = stoi(row[0]);

// Compare the roll number
if (roll2 == rollnum) {

    // Print the found data
    count = 1;
    cout << "Details of Roll " << row[0] << " : \n";
    cout << "Name: " << row[1] << "\n";
    cout << "Maths: " << row[2] << "\n";
    cout << "Physics: " << row[3] << "\n";
    cout << "Chemistry: " << row[4] << "\n";
    cout << "Biology: " << row[5] << "\n";
    break;
}
if (count == 0)
    cout << "Record not found\n";
}

```

### Output:

```

OUTPUT PROBLEMS DEBUG CONSOLE TERMINAL

nayonika@nayonika-HP-Notebook:~/Documents/cpp lab/cpp project/profinal$ ./a.out
Enter the roll number of the student to display details: 2
Details of Roll 2 :
Name: kaushik
Maths: 90
Physics: 80
Chemistry: 50
Biology: 90
nayonika@nayonika-HP-Notebook:~/Documents/cpp lab/cpp project/profinal$ ./a.out
Enter the roll number of the student to display details: 8
Record not found
nayonika@nayonika-HP-Notebook:~/Documents/cpp lab/cpp project/profinal$ 

```

### 3. Update a record:

The following approach is implemented while updating a record:-

1. Read data from a file and compare it with the user input, as explained under read operation.
2. Ask the user to enter new values for the record to be updated.
3. update row[index] with the new data. Here, index refers to the required column field that is to be updated.
4. Write the updated record and all other records into a new file ('reportcardnew.csv').
5. At the end of operation, remove the old file and rename the new file, with the old file name, i.e. remove 'reportcard.csv' and rename 'reportcardnew.csv' with 'reportcard.csv'

## UPDATE

```
void update_recode()
{
    // File pointer
    fstream fin, fout;

    // Open an existing record
    fin.open("reportcard.csv", ios::in);

    // Create a new file to store updated data
    fout.open("reportcardnew.csv", ios::out);

    int rollnum, roll1, marks, count = 0, i;
    char sub;
    int index, new_marks;
    string line, word;
    vector<string> row;

    // Get the roll number from the user
    cout << "Enter the roll number "
        << "of the record to be updated: ";
    cin >> rollnum;

    // Get the data to be updated
    cout << "Enter the subject "
        << "to be updated(M/P/C/B): ";
    cin >> sub;

    // Determine the index of the subject
    // where Maths has index 2,
    // Physics has index 3, and so on
    if (sub == 'm' || sub == 'M')
        index = 2;
    else if (sub == 'p' || sub == 'P')
        index = 3;
    else if (sub == 'c' || sub == 'C')
        index = 4;
    else if (sub == 'b' || sub == 'B')
        index = 5;
    else {
        cout << "Wrong choice.Enter again\n";
        update_record();
    }

    // Get the new marks
    cout << "Enter new marks: ";
    cin >> new_marks;

    // Traverse the file
    while (!fin.eof()) {

        row.clear();
```

```
getline(fin, line);
stringstream s(line);

while (getline(s, word, ',', ',')) {
    row.push_back(word);
}

roll1 = stoi(row[0]);
int row_size = row.size();

if (roll1 == rollnum) {
    count = 1;
    stringstream convert;

    // sending a number as a stream into output string
    convert << new_marks;

    // the str() converts number into string
    row[index] = convert.str();

    if (!fin.eof()) {
        for (i = 0; i < row_size - 1; i++) {

            // write the updated data
            // into a new file 'reportcardnew.csv'
            // using fout
            fout << row[i] << ", ";
        }

        fout << row[row_size - 1] << "\n";
    }
}

else {
    if (!fin.eof()) {
        for (i = 0; i < row_size - 1; i++) {

            // writing other existing records
            // into the new file using fout.
            fout << row[i] << ", ";
        }

        // the last column data ends with a '\n'
        fout << row[row_size - 1] << "\n";
    }
}

if (fin.eof())
    break;
}

if (count == 0)
    cout << "Record not found\n";

fin.close();
fout.close();

// removing the existing file
remove("reportcard.csv");
```

```
// renaming the updated file with the existing file name
rename("reportcardnew.csv", "reportcard.csv");
}
```

**Output:**

The terminal window shows the contents of the CSV file 'reportcard.csv' with the following data:

Roll Number	Name	Marks
1	rahul	100, 90, 75, 100
2	kaushik	90, 80, 50, 90
3	nayonika	90, 78, 95, 86
4	simran	55, 85, 70, 70
5	sumana	65, 90, 95, 100
6	shiv	78, 75, 84, 85
7		

Below the file content, a command-line session is shown:

```
nayonika@nayonika-HP-Notebook:~/Documents/cpp_lab/cpp project/profinal$ ./a.out
Enter the roll number of the record to be updated: 3
Enter the subject to be updated(M/P/C/B): p
Enter new marks: 78
nayonika@nayonika-HP-Notebook:~/Documents/cpp_lab/cpp project/profinal$
```

**4. Delete a record:**

The following approach is implemented while deleting a record

1. Read data from a file and compare it with the user input, as explained under read and update operation.
2. Write all the updated records, except the data to be deleted, onto a new file (reportcardnew.csv).
3. Remove the old file, and rename the new file, with the old file's name.

**DELETE**

```
void delete_record()
{
    // Open FILE pointers
    fstream fin, fout;

    // Open the existing file
    fin.open("reportcard.csv", ios::in);

    // Create a new file to store the non-deleted data
    fout.open("reportcardnew.csv", ios::out);

    int rollnum, roll1, marks, count = 0, i;
    char sub;
    int index, new_marks;
    string line, word;
    vector<string> row;

    // Get the roll number
    // to decide the data to be deleted
```

```

cout << "Enter the roll number "
     << "of the record to be deleted: ";
cin >> rollnum;

// Check if this record exists
// If exists, leave it and
// add all other data to the new file
while (!fin.eof()) {

    row.clear();
    getline(fin, line);
    stringstream s(line);

    while (getline(s, word, ',', ',')) {
        row.push_back(word);
    }

    int row_size = row.size();
    roll1 = stoi(row[0]);

    // writing all records,
    // except the record to be deleted,
    // into the new file 'reportcardnew.csv'
    // using fout pointer
    if (roll1 != rollnum) {
        if (!fin.eof()) {
            for (i = 0; i < row_size - 1; i++) {
                fout << row[i] << ", ";
            }
            fout << row[row_size - 1] << "\n";
        }
    }
    else {
        count = 1;
    }
    if (fin.eof())
        break;
}
if (count == 1)
    cout << "Record deleted\n";
else
    cout << "Record not found\n";

// Close the pointers
fin.close();
fout.close();

// removing the existing file
remove("reportcard.csv");

// renaming the new file with the existing file name
rename("reportcardnew.csv", "reportcard.csv");
}

```

## Output:

```

gfg.cpp reportcard.csv *
1 1,rahul,100,90,75,100
2 2,kaushik,90,80,50,90
3 3,nayonika,90,78,95,86
4 5,sumana,65,90,95,100
5 6,shiv,78,75,84,85
6

OUTPUT PROBLEMS DEBUG CONSOLE TERMINAL

nayonika@nayonika-HP-Notebook:~/Documents/cpp_lab/cpp project/profinal$ ./a.out
Enter the roll number of the record to be deleted: 4
Record deleted
nayonika@nayonika-HP-Notebook:~/Documents/cpp_lab/cpp project/profinal$ 

```

**References:** <https://www.geeksforgeeks.org/file-handling-c-classes/>,

<https://www.geeksforgeeks.org/stringstream-c-applications/>

30

## Related Articles

1. Bookshop management system using file handling

---

2. C program to copy contents of one file to another file

---

3. How to create Binary File from the existing Text File?

---

4. C++ Program to Copy One File into Another File

---

5. C++ Program to Read Content From One File and Write it Into Another File

---

6. C++ Program to Copy the Contents of One File Into Another File

---

7. Difference Between C++ Text File and Binary File

---

8. ATM Management System using C++

---

9. Student record management system using linked list

---

10. Department Store Management System(DSMS) using C++

Previous

Next

**SALE!****GeeksforGeeks Courses Upto 25% Off Enroll Now!****Save 25% on Courses**

DSA

Data Structures

Algorithms

Interview Preparation

Data Science

T

# Four File Handling Hacks which every C/C++ Programmer should know

Difficulty Level : Medium • Last Updated : 15 Mar, 2023

**Read****Discuss****Courses****Practice****Video**

We will discuss about four file hacks listed as below-

1. Rename – Rename a file using C/C++
2. Remove – Remove a file using C/C++
3. File Size – Get the file size using C/C++
4. Check existence – Check whether a file exists or not in C/C++

## C++

```
// C++ Program to demonstrate the
// four file hacks every C/C++ must know

// Note that we are assuming that the files
// are present in the same file as the program
// before doing the below four hacks

#include <iostream>
#include <stdlib.h>
using namespace std;

// A Function to get the file size
unsigned long long int fileSize(const char *filename)
{
    // Open the file
    FILE *fh = fopen(filename, "rb");
    fseek(fh, 0, SEEK_END);
    unsigned long long int size = ftell(fh);
    fclose(fh);

    return size;
}
```

```

// A Function to check if the file exists or not
bool fileExists(const char * fname)
{
    FILE *file;
    if (file = fopen(fname, "r"))
    {
        fclose(file);
        return true;
    }
    return false;
}

// Driver Program to test above functions
int main()
{
    cout << fileSize("Passwords.txt") << " Bytes\n";
    cout << fileSize("Notes.docx") << " Bytes\n";

    if (fileExists("OldData.txt") == true )
        cout << "The File exists\n";
    else
        cout << "The File doesn't exist\n";

    rename("Videos", "English_Videos");
    rename("Songs", "English_Songs");

    remove("OldData.txt");
    remove("Notes.docx");

    if (fileExists("OldData.txt") == true )
        cout << "The File exists\n";
    else
        cout << "The File doesn't exist\n";

    return 0;
}

// This code is contributed by sarajadhav12052009

```

## C

```

// A C Program to demonstrate the
// four file hacks every C/C++ must know

// Note that we are assuming that the files
// are present in the same file as the program
// before doing the below four hacks
#include<stdio.h>
#include<stdlib.h>
#include<stdbool.h>

// A Function to get the file size
unsigned long long int fileSize(const char *filename)
{

```

```

// Open the file
FILE *fh = fopen(filename, "rb");
fseek(fh, 0, SEEK_END);
unsigned long long int size = ftell(fh);
fclose(fh);

return (size);
}

// A Function to check if the file exists or not
bool fileExists(const char * fname)
{
    FILE *file;
    if (file = fopen(fname, "r"))
    {
        fclose(file);
        return (true);
    }
    return (false);
}

// Driver Program to test above functions
int main()
{
    printf("%llu Bytes\n", fileSize("Passwords.txt"));
    printf("%llu Bytes\n", fileSize("Notes.docx"));

    if (fileExists("OldData.txt") == true )
        printf("The File exists\n");
    else
        printf("The File doesn't exist\n");

    rename("Videos", "English_Videos");
    rename("Songs", "English_Songs");

    remove("OldData.txt");
    remove("Notes.docx");

    if (fileExists("OldData.txt") == true )
        printf("The File exists\n");
    else
        printf("The File doesn't exist\n");

    return 0;
}

```

Output:

```
C:\Users\BELWARIAR\Desktop\Test\Four_File_Hacks.exe
14 Bytes
9924 Bytes
The File exists
The File doesn't exist

-----
Process exited after 0.09681 seconds with return value 0
Press any key to continue . . .
```

**Screenshot before executing the program :**

AD

Name	Date modified	Type	Size
PythonPrograms	19-06-2016 08:09	File folder	
Songs	19-06-2016 08:07	File folder	
Videos	19-06-2016 08:07	File folder	
Four_File_Hacks	19-06-2016 08:19	C++ Source File	2 KB
Notes	19-06-2016 08:19	Microsoft Office ...	10 KB
OldData	19-06-2016 08:08	Notepad++ Docu...	1 KB
Passwords	19-06-2016 08:10	Notepad++ Docu...	1 KB

**Screenshot after executing the program :**

Name	Date modified	Type	Size
English_Songs	19-06-2016 08:07	File folder	
English_Videos	19-06-2016 08:07	File folder	
PythonPrograms	19-06-2016 08:09	File folder	
Four_File_Hacks	19-06-2016 08:33	C++ Source File	2 KB
Passwords	19-06-2016 08:10	Notepad++ Docu...	1 KB

This article is contributed by **Rachit Belwariar**. If you like GeeksforGeeks and would like to contribute, you can also write an article and mail your article to review-team@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

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

14

## Related Articles

1. Bitwise Hacks for Competitive Programming

---

2. 10 C++ Programming Tricks That You Should Know

---

3. C | File Handling | Question 1

---

4. C | File Handling | Question 2

---

5. C | File Handling | Question 3

---

6. C | File Handling | Question 4

---

7. C | File Handling | Question 5

---

8. Set Position with seekg() in C++ File Handling